

AD-A111 566

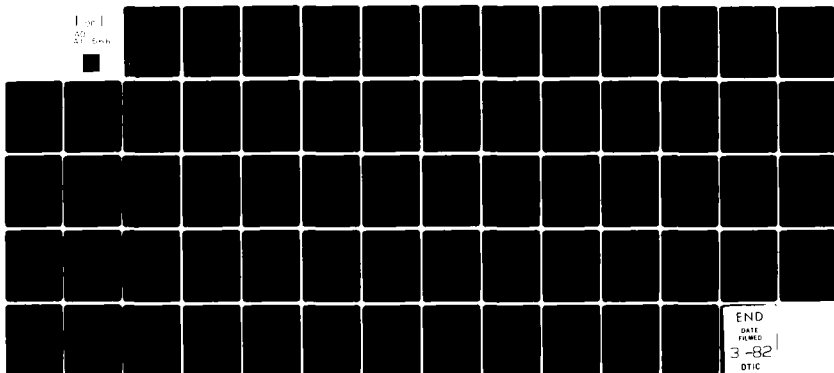
FORD AEROSPACE AND COMMUNICATIONS CORP PALO ALTO CA W--ETC F/G 17/2
KSOS FINAL REPORT (KERNELIZED SECURE OPERATING SYSTEM). (U)
AUG 81

MDA903-77-C-0333

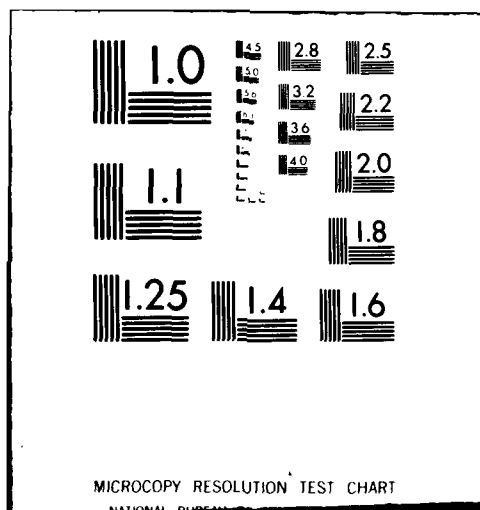
NL

UNCLASSIFIED

For
20 6-81



END
DATE
FILMED
3-82
DTIC



ADA111566

②

SECURE MINICOMPUTER OPERATING SYSTEM (KSOS)
FINAL REPORT

Department of Defense Kernelized Secure Operating System

Contract MDA 903-77-C-0333
CDRL 0002BG


DTIC
SELECTED
MAR 03 1982
S E D

Prepared for:

Defense Supply Service-Washington
Room 1D245, The Pentagon
Washington, D.C. 20310

DTIC FILE COPY

Approved for public release; distribution unlimited.


Ford Aerospace &
Communications Corporation
Western Development
Laboratories Division

Walter Fabian Way
84303

82 03 03 030

CONTENTS

1. INTRODUCTION	1
1.1 Scope	1
1.2 Organization	1
2. EXECUTIVE SUMMARY	2
2.1 Introduction	2
2.2 Project Goals and Their Realization	2
2.2.1 Security Assurance	2
2.2.2 UNIX Compatability	3
2.2.3 Efficiency	3
2.2.4 Broad Applicability	3
2.3 Problems Along the Way	5
2.3.1 Consistency Between Representations	5
2.3.2 Multiple Language Support	5
2.3.3 Modula	5
2.3.4 Formal Tools	6
2.3.5 Mathematical Model Limitations	7
2.4 Conclusion	7
3. HISTORICAL OVERVIEW	8
3.1 The Need	8
3.2 The Security Kernel	8
3.3 The Security Policy Model	9
3.4 Mathematical Techniques Required	10
3.5 Early Experiments	10
3.6 KSOS Beginning	10
4. GOALS, OBJECTIVES, and REQUIREMENTS	12
4.1 Goal	12
4.2 Objectives	12
4.3 Requirements	12
4.3.1 Structural Requirements	12
4.3.2 Formal Security Requirements	13
4.3.3 Development Methodology Requirements	13
4.3.4 Verification Requirements	13
4.3.5 Usability Requirements	13
5. DEVELOPMENT	16
5.1 The Kernel	19
5.1.1 Introduction	20
5.1.2 Kernel I/O	20
5.1.3 Kernel Process Management	21
5.1.4 Kernel Segment Management	23
5.1.5 Conclusion	24
5.2 Emulator	24
5.2.1 Introduction	24
5.2.2 Runtime Executive	24
5.2.3 I/O System	24
5.2.4 Process System	24
5.2.5 Miscellaneous Services	24
5.2.6 TCP Services	24
5.2.7 Testing and Integration	25
5.2.8 Conclusion	25
5.3 NKS	25
5.3.1 Introduction	25

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	



5.3.2 Technical Development Experience	27
5.3.3 Security Experience	28
5.3.4 Conclusion	30
5.4 Network	30
5.4.1 Approach	30
5.4.2 Protocol	33
5.4.3 Implementation Problems	34
5.4.4 Postscript	34
6. METHODOLOGY	35
6.1 Languages	35
6.1.1 Implementation Language Selection	35
6.1.2 Hierarchical Design Language (HDL)	39
6.2 Hierarchical Development Methodology (HDM)	40
6.2.1 Assumptions	40
6.2.2 HDM Specifications	40
6.3 Verification	41
6.3.1 Introduction	42
6.3.2 KSOS Verification Achievements	42
6.3.3 Difficulties	53
6.3.4 Conclusions	54
6.4 Testing	54
6.4.1 Unit Testing	54
6.4.2 Integration Testing	55
6.4.3 Emulator Qualification Testing	55
6.4.4 NKSR CPCI Qualification Testing	56
6.4.5 Kernel Qualification Testing	56
6.4.6 System Testing	57
6.4.7 Conclusion	57
7. RESULTS	58
8. ACKNOWLEDGMENTS	60
9. REFERENCE DOCUMENTS	61
9.1 KSOS Documents	61
9.2 Other Documents	62

KSOS Final Report

1. INTRODUCTION

1.1 Scope

This document, the Kernelized Secure Operating System (KSOS) Final Report, describes work performed by Ford Aerospace & Communications Corporation (FACC) and its subcontractor SRI International, under ARPA order 3319 contract MDA903-77-C-0333, issued by Defense Supply Service—Washington in response to Proposal WDL-TP8110, furnished in connection with Request for Proposals MDA903-77-R-0077. The work was funded by various DoD agencies. It contains a description of the goals, evolution, and results of the KSOS program, an introduction to the structure of KSOS, a description of the approach, an exposition of the accomplishments, and a review of experience gained from participation in the program.

1.2 Organization

The remaining sections of this report are organized as follows:

- Section 2 is an executive summary describing the goals, results, and conclusions of the KSOS program in summary form.
- Section 3 is an historical overview of the KSOS program, the preceding events and experiments. In addition, the motivations behind KSOS are discussed and some fundamental aspects of the project are introduced.
- Section 4 presents the goals, objectives, and original requirements for the KSOS system.
- Section 5 is a detailed discussion of the design and development of the various parts of the KSOS software.
- Section 6 deals with the methodology for KSOS development including: programming language selection and experience, Hierarchical Development Methodology (HDM) experience, formal verification, and testing.
- Section 7 describes the results of the project and the KSOS system as delivered.
- Sections 8 and 9 are the Acknowledgments and References respectively.

KSOS Final Report

2. EXECUTIVE SUMMARY

2.1 Introduction

The Kernelized Secure Operating System (KSOS) was developed by Ford Aerospace and Communications Corporation (FACC) as a provably secure replacement for the UNIX¹ operating system. KSOS will run on an unmodified Digital Equipment Corporation PDP-11/70² computer system. A similar project is underway within the Honeywell Corporation to provide a KSOS-like system for the SCOMP, a modified Level 6 computer system. (Where confusion might arise, KSOS-11 and KSOS-6 are used for the FACC and Honeywell efforts respectively.)

The KSOS project got underway in 1977 with the award of two competitive design contracts. After eight months of design, FACC was selected to continue into implementation.

KSOS is composed of three major pieces. The Security Kernel provides a primitive, secure operating system. The UNIX Emulator is part of every UNIX process which maps the interface provided by the Kernel into one essentially identical to that provided by the UNIX operating system. The third component is a collection of autonomous programs referred to as the Non-Kernel, Security-Related Software (NKSr). The NKSr includes both trusted and untrusted processes. Trusted processes have the privilege to violate one or more of the Kernel's rules or to issue privileged Kernel calls, untrusted processes do not. An example of an untrusted NKSr process would be the editor used by the system administrator to modify the data base containing user security clearance information. This process need not violate any Kernel rules, but it is crucial to system security.

2.2 Project Goals and Their Realization

The goals of the KSOS project were to build a system that was:

- a. secure, and capable of having mathematically sound statements made about its security,
- b. compatible with UNIX, including comparable efficiency,
- c. suitable for a broad range of applications.

In attempting to realize these goals many insights have been gained.

2.2.1 Security Assurance

The security of KSOS comes from three distinct aspects of the development of the system. The first is through the use of formal methodology. KSOS has attempted to add mathematical rigor to many different stages of the project. Formal specifications of the trusted parts of the system have been written. The specifications of the Kernel interface have been mechanically proven to comply with a mathematical model of DoD security policy. Tools have been developed to prove correspondence between the formal specifications and the implementation of components of the system in a high level language (Modula). For budgetary reasons only trivial demonstrations of the correspondence proof process have been completed. Much of the system is written in Modula, a language supporting strong typing of variables and language enforced modularity.

Security assurance in KSOS also comes from the disciplined application of software engineering principles. The development environment for KSOS is a heavily supplemented version of PWB/UNIX. It provides an integrated, on-line software development environment in which dramatic improvements in staff productivity are possible. A comprehensive configuration management system allows changes to software, documentation, etc. to be traced and controlled. A project library system has become the single repository for all project related information including source code, documents, trouble reports, technical notes, etc.

Finally, security of the system comes from testing. Although the KSOS project did not reach the point of performing formal tests, KSOS has pioneered the development of test cases from the formal specifications for the system. In this way, it is possible to demonstrate that the realization of the system as object code complies with its formal specifications. The specifications, in turn, have been proven to comply with a mathematical model of DoD security policy. While formal testing is not as theoretically sound as proofs of

1. UNIX and PWB/UNIX are trademarks of the Bell System.

2. PDP and DEC are trademarks of the Digital Equipment Corporation.

KSOS Final Report

correspondence between the code and the specifications, it does provide additional confidence in the correct operation of the system. Formal testing can also be both more complete and more economical than conventional, ad hoc test case generation techniques.

2.2.2 UNIX Compatability

The goal of UNIX compatibility means that binary images of existing programs which operate on the UNIX operating system will operate correctly on KSOS, providing that the program neither contains inherent security flaws, nor takes advantage of idiosyncrasies and undocumented features of particular versions of UNIX. There is a vast and growing set of programs for the UNIX operating system. The intent of this goal was to preserve this applications software base. The KSOS project has made diligent efforts to meet this goal. In some cases, attempting to meet the goal introduced additional complexity into portions of the system.

2.2.3 Efficiency

In addition to the semantic compatibility with UNIX, KSOS was to offer comparable efficiency. The goal was that KSOS would operate no worse than a factor of two slower than a comparable UNIX system performing the same workload. This goal has not been met for several now apparent reasons:

- a. Security costs something; additional checks must be made, implementation structures are often chosen more from security considerations than from efficiency considerations.
- b. A given UNIX operating system call may require processing both in the UNIX Emulator and the Security Kernel, incurring additional overhead.
- c. UNIX is a very mature system. Over its life many different structural alternatives have been explored and the most effective have survived. Although the primary goal of UNIX has always been simplicity, it has been remarkably successful in achieving efficiency too. KSOS is just beginning its life, and has not yet reached similar maturity.
- d. Due to a desire to minimize the size of the Kernel, some functionality has been moved to the Emulator or to NKSIR processes. This structuring reduces the opportunity for global optimization of various decisions, such as disk accessing and buffering.

In attempting to meet the efficiency goal, the project has made several design decisions which differ from the prototype efforts which preceded KSOS. The most important difference is that KSOS has a more monolithic Kernel instead of distributing functions out to "symbiotic" trusted processes. This choice was made because of high process switching costs on the PDP-11/70. The KSOS Kernel is a primitive, yet functionally complete, operating system. This has made the KSOS Kernel larger than its predecessors.

A second difference is that KSOS is a swapping system instead of supporting demand paging. Analysis of UNIX software by the project and limited experiments done at the Naval Postgraduate School suggest that the working set of typical UNIX system software comprises nearly all of the pages of the program. That is, the program can be expected to reference nearly all of its pages within a short time from any given instant. Further supporting the decision to not support demand paging was the rather large (8 K bytes) size of usable PDP-11/70 pages. (Although the memory management unit can support smaller pages, the virtual memory cannot be completely spanned with pages less than 8 K bytes.)

It is interesting to note that the new machine architectures offered by several vendors remedy nearly all of these limitations of the PDP-11/70.

2.2.4 Broad Applicability

The KSOS system was to be more than a proof of concept. Rather, it was intended to serve as the basis for projects requiring a trusted system. Thus, KSOS had to incorporate features which enhanced its applicability to a diverse set of projects. These features can be divided into two major categories:

- a. administrative support functions
- b. reductions in the UNIX dependencies of the system

KSOS Final Report

2.2.4.1 Administrative Support

The distributed Version 6.0 UNIX system contains very little in the way of administrative support features. The file system is vulnerable to certain machine failures. Repair may require a "UNIX-guru" to restore logical consistency. This is in keeping with another fundamental tenet of UNIX design, that it is assumed that many of the users of the system are also well qualified to make modifications to the operating system and utility programs. KSOS was designed from a different premise. While a thoroughly knowledgeable and well equipped maintenance organization is assumed for KSOS, individual installations were assumed to be operated in a more turnkey mode. Thus, there had to be more support provided for the administrative operation of the system by comparatively naive users. System security considerations necessitated other new pieces of software.

For example, it was undesirable to use the standard editors for management of the system's security control data bases, such as the file containing the users' identities and their security clearances. One could posit scenarios in which a standard editor is subtly modified to unobtrusively alter such a data base if the editor is ever used by the administrator in an otherwise legitimate way. This scenario is an example of a "trojan horse" penetration of the system. To guard against such penetrations, specialized editors for these data bases have been created.

Many operating systems have an all-powerful user who is not subject to the system security rules. When operating as this "superuser", any file can be read or written and any process can be killed, among other things. In a trusted system, such an untrammelled capability is unacceptable. KSOS has adopted a different philosophy. Instead of having an all-powerful user who makes use of ordinary programs in extraordinary ways, in KSOS there are specific privileges associated with particular executable image (program) files. Thus, a particular function is encapsulated by providing a privileged program that will perform that function and no more. There are further access restrictions imposed on these image files. For example, a normal user can invoke privileged functions to change his/her current working security level (e.g. "I was working at the Secret level, now let me work at the Unclassified level."). Other functions such as dumping a file system to tape or initiating the network service routines are available only to the operator. Finally, still other functions such as installing a new user or altering the clearances of an existing user are available only to the system administrator.

The unanswered question at this point is whether the ensemble of these services is sufficient to operate the system. In FACC's experience with over fifteen in-house UNIX systems, the planned KSOS utility suite would have been adequate for all known instances of system administration and crash repair. If the present suite is inadequate, adding new encapsulated functions would be quite easy.

2.2.4.2 Reduced UNIX Dependencies

Many applications for KSOS do not require UNIX, but are suitable for implementation directly on the Kernel. Thus, it is desirable that the Kernel not contain a great many UNIX-specific features which would hinder or complicate such applications. The design of the Kernel attempted to provide good support for UNIX while not being specific to it. Evidence of the success of these efforts can be found in several areas:

- a. A large amount of KSOS software runs directly on the Kernel. Because the UNIX Emulator is not subject to as much scrutiny as the Kernel, any program which has privileges to violate the security rules of the system must operate directly on the Kernel. Writing such software required only some libraries to create an acceptable programming environment. The existence of this software demonstrates that it is possible to build Kernel applications that do not use the Emulator.
- b. The design of KSOS-6, the Honeywell project, was based on KSOS-11. Although there are substantial differences due to a much different hardware base, KSOS-6 and KSOS-11 share many architectural similarities.
- c. Studies of the feasibility of designing an emulator for DEC's RSX-11M Plus to run on a modified KSOS Kernel have suggested that only minor modifications to the Kernel would be required. The RSX-11M Plus Emulator would support a proper, though nearly complete, subset of RSX-11M Plus features.

It is an interesting speculation whether kernel/emulator technology might be an attractive possibility for the support of multiple operating systems for the same hardware base.

2.3 Problems Along the Way

The KSOS project has faced and overcome many problems in realizing these goals. Nearly all of the problems stem from attempting to move technology from the research arena into production use.

2.3.1 Consistency Between Representations

Perhaps the most deep-seated issue is that of maintaining consistency between different representations of the system. For example, the functioning of a given Kernel call might be described in as many as seven different representations. In classical software projects the problem is less severe because only one representation is definitive, the code. All of the other representations could be made to conform to the code. In KSOS the code and the formal specifications for it are, or should be, equally definitive. In the long run, these consistency problems could be managed through proofs of correspondence. In the KSOS project and for the near term future, such correspondence checking can only be accomplished through diligent manual analysis.

There is a semi-independent path for checking the adequacy of the manual analysis, that of the formal testing described earlier. Because the test cases are generated by an independent test team from the formal specifications with little or no reference to the code which realizes the specifications, there is some check on the success of the consistency control process. Although this is a primarily managerial solution to an essentially technical problem, it does rest upon on some theoretically sound underpinnings. It should be noted that minor differences in specification style can have dramatic impact on the ease with which a formal specification may be checked against its realization or other supporting documentation.

2.3.2 Multiple Language Support

Related to the consistency management problem was the need to support the various languages used on the project. Members of the project had to become fluent in several programming languages (Modula, C, assembler, PWB Shell, etc.) as well as special purpose languages for text processing, formal specification and software design.

Over the course of the project every one of the language processors used required modification, ranging from trivial increases in table size parameters to creation of brand new processors.

The selection and enforcement of standards for the use of the various languages presented challenges. For the programming languages, standards definition and adherence were fairly easy, since there were many examples to draw upon. There were some problems in selection of how constructs in SPECIAL, the formal specification language used, were to be represented in Modula. Particularly troublesome was the representation of EXCEPTIONS. Exactly how these were to be represented in Modula was a controversy whose resolution has never been universally acceptable. Standards for the use of a structured English design language were much more trouble. The project started with very exacting standards which imposed a form of strong typing on the design. This proved unworkable because it forced vastly too much detail into the design, obscuring the essential aspects. Because the design language did not match closely with Modula in such areas as scoping rules and iteration constructs, the mapping from design to code was more abrupt than would be desirable. Other standards for the project met with greater success, such as the directory topology and the file naming conventions for the project library. Standard techniques for the construction of software from its components were developed and have been very successful.

2.3.3 Modula

The Modula compiler problems were severe enough to deserve special mention. Modula was chosen because of its great elegance and simplicity, both of which are crucial for any serious attempts at formal proofs of correspondence between specifications and code. The problems stemmed from attempting to take a successful research product and apply it in a much larger case than it had ever been used before. Modula did not support separate compilation. That is, the entire Kernel had to be compiled at once. When Modula was selected, separate compilation was viewed as primarily a convenience item of minor significance. This was not the case. Nearly every point in the compilation, assembly, and loading path eventually turned out to be a bottleneck which had to be enlarged, often with major surgery, to accommodate the entire Kernel. The project received invaluable help from the University of York (England) in solving these problems.

KSOS Final Report

Modula also contributed to another problem, the size of the executable image of the Kernel. When the language was selected, one of the criteria was efficiency of the resulting object code. Modula provided some improvements over the competition in several areas, the most important being procedure call/return sequences. Modula encourages data abstraction in which data structures are accessed via procedures instead of directly. Even though the Modula procedure call is fairly cheap, it is still much more expensive than directly accessing the data structure. In one instance, making a structure global saved several hundred bytes of instructions. Modula also lacks pointers, so there are many instances of array offset calculations. Finally, the language lacks some of the richness of C, the other commonly used language on KSOS, in the area of bit manipulation and support for 32 bit arithmetic. All of these seemingly minor factors cascaded to make the executable image for the Kernel much larger than expected. This meant continual efforts to pare down the size of the Kernel.

Mitigating these difficulties are the substantial benefits gained from the strong typing, language-generic multiprogramming and enforced modularity of Modula. If the language selection process were to be repeated today, it is likely that Modula would still be a very strong contender.

2.3.4 Formal Tools

The tools used to state and prove things about the system also posed operational problems. SPECIAL is one of the first attempts at a formal specification language. It is a tribute to its designers that it has been so successful. However, it is generally agreed that SPECIAL has limitations which must be overcome in subsequent efforts. The most fundamental of these limitations is the underlying computational model. SPECIAL cannot express certain multi-process interactions in a clean way. For most of the interface to an operating system this is not a problem. However, there are instances in which meaningful system semantics cannot be formally described. Minor "glitches" in the language syntax or semantics create other small, but annoying, inability to express desired relationships. SPECIAL by itself cannot express complex sequences of actions in which the flow is dependent upon the success or failure of prior steps. Such situations are common in the specification of trusted processes running on the Kernel. It is anticipated that current efforts in methodology research as well as a more rigorous application of the other aspects of the Hierarchical Development Methodology will alleviate this problem in future work.

- The tools for proving the security properties of SPECIAL specifications are a major success for KSOS.
- Briefly, a large number (approximately 1800) of candidate theorems are generated from the formal specifications. These theorems relate the security level of objects accessed by the process to the level of the process. If all of the candidate theorems can be proven, then the specification satisfies the model of DoD security policy. The process of generation and proof is entirely automated, requiring no human intervention after initial setup. Also, the speed of the process was increased by several orders of magnitude during the course of the project. This meant that the specifications could be cycled through the prover approximately once per day. Due to the extreme computational load imposed by the process of generation and proof, it was impractical to run it during the day, so a batch job was submitted to run in the middle of the night. The ability to make changes and to try again was extremely valuable.

Because the theorem generation process takes a low level view of the specifications, it generates some candidate theorems which are actually false (i.e. an apparent security flaw) but in which the apparent flaw is prevented by higher level semantics.

The theorem generation and proof process has uncovered several very subtle security flaws. This is particularly impressive given the maturity of the specifications, and the great deal of effort which went into their creation in attempting to prevent the inclusion of security flaws.

It has become clear that it is very desirable to make proofs of other properties of the specifications. Exactly what needs to be proven and the limitations of such a process are a research area of potentially great importance.

The tools to prove correspondence between Modula programs and their formal specifications were not so successful. This is a fundamentally difficult problem which is exacerbated by a lack of experience with real systems. To date, the KSOS project has accomplished laying of groundwork which can be built upon in future code proof efforts. Although only very simple code proofs have been accomplished so far, the prospect of substantial code proof efforts remains attractive. Even though not very many were attempted, preparing to do code proofs has paid benefits, since languages and system structures amenable to code proof

KSOS Final Report

facilitate debugging and modification.

2.3.5 Mathematical Model Limitations

The mathematical model of DoD security policy turned out to be inadequate for describing a real system. A real system, for reasons of operational viability, requires certain support and maintenance functions that violate the model. In a strict interpretation of the model, such a violation invalidates statements about the security of subsequent states of the system. The model relies upon starting the system in a secure state and restricting system state transitions to ones which preserve the security of the system. KSOS and other trusted systems have made engineering tradeoffs in providing essential services which inherently violate the security model. For example, the rate at which users can cause a trusted service to violate the security model is limited by artificially inserted delays. Although quite defensible as being analogous to the way in which the paper security world operates, the inability to stay within the model is disturbing. What is probably needed is a more comprehensive model in which the existing model could be incorporated. Also needed are unifying principles regarding the correctness of trusted software. Presently, it is quite difficult to define exactly what "correct" means for a trusted process. Each instance must be treated separately. Finally, there needs to be more formalization and proof of the other protection mechanisms found in systems like KSOS. The discretionary access (i.e. protection defined by the owner of the object) and encapsulation rules both need formalization. Once such formalization is complete, proofs that the specifications satisfy the required properties can be attempted.

2.4 Conclusion

The preceding sections have discussed some of the lessons learned in the course of the KSOS project. Obviously there are also a number of other detailed technical and managerial lessons learned. Of much greater importance, perhaps, are the following results:

- success of the basic design principles,
- transfer of tools and techniques from the research community to production use,
- and a demonstration that such systems can be built.

Despite the length of the project and the wealth of detailed experience gained regarding the viability of the assumptions upon which the system was conceived, the basic design principles have been lasting and successful. While packaging, implementation details, and some minor semantics have changed, the basic decomposition of the system has remained fixed.

The tools and techniques employed in the construction of the system have been moved from the research community to production use. Occasionally, the transition has been far more painful than anyone expected, demonstrating that there is still much interesting research work left to do.

The importance of KSOS goes beyond its immediate existence. What is far more important is the demonstration that such systems can be built. There is increasing interest from vendors about the incorporation of stronger security assurance into their products. While KSOS has had its share of problems, it has shown the viability of many concepts.

KSOS Final Report

3. HISTORICAL OVERVIEW

3.1 The Need

In the early 1970's various groups within the DoD/Intelligence community realized that the computerized information systems coming into widespread use posed serious security risks. The available operating systems could invariably be penetrated by "tiger teams", giving them the ability to read or modify any information in the system. The security risks inherent in existing and proposed systems forced the community to utilize external measures to provide the required system security. These measures included dedication of systems to classified information processing tasks, physical security measures to encapsulate the system, and "periods processing", that is, running in a dedicated mode for some period of time and purging classified information before and after the periods.

These external measures were unsatisfactory for many reasons. First, they often posed unacceptable operational constraints. Information which originated in a classified environment had to be regarded as classified at "system high" (the highest security level processed by the system) until manually examined. This was particularly inconvenient when only a small amount of information was highly classified. Because the system could not be trusted not to mix information, external review of supposedly unclassified material was required. The need for dedicated systems to process classified information led to duplication of information and the possibility of de-synchronization between the classified and unclassified copies.

A second class of problem was economic. Dedication of systems prevents load sharing and effective utilization of resources. Purging the system between periods of processing at different security levels meant many hours of lost time per day.

Finally, as increasingly integrated environments were contemplated, it became more difficult to have all individuals having access to the system cleared to system high. Even if they could all be cleared, such practices violated the need to know principle which is a foundation of security policy.

Initially, some members of the community felt that contemporary systems could be "hardened" to make them less vulnerable to attempted penetrations. Studies were undertaken to categorize the types of security flaws found in systems and to suggest mechanisms for eliminating them. However, as such efforts progressed it became increasingly clear that security could not practically be retrofitted into a system. Existing operating systems were massively large and extremely complicated programs. There were too many places where a minor error in design or implementation could be exploited as a security flaw.

These failures at security retrofitting led to a realization that more basic research was needed in computer security. The most promising avenues of research seemed to lie in the area of program verification. The hope was that the power of mathematics could be harnessed to assure the correct implementation of a system.

3.2 The Security Kernel

In response to the shortcomings of existing systems a panel of experts was convened by the community to set down the basic principles which were to guide computer security research through the present day. Their recommendations presented in the "Ware Report" (after its editor) suggested an overall system architecture which, with but minor variation, has been followed by all secure systems to date.

The panel recognized that many of the security flaws in systems could be traced to the size and complexity of then current operating systems. A common paradigm was that in all but one case the conventions expected by a particular routine were obeyed. The one slight inconsistency could be exploited into a penetration of the system. To reduce the chances for such errors the panel suggested an alternative architecture. The system would be decomposed in a different way. All of the security critical features would be placed in a small component of the system, the Security Kernel. To improve the possibility of rigorous examination, the Security Kernel was to be kept as small as possible. The Kernel would satisfy three basic properties:

- it would be complete. Every access attempt would be mediated by the Kernel. To produce a system with acceptable performance, some of these checks would be made by hardware, e.g. memory management hardware checking the legitimacy of every memory reference.
- it would be correct. the Kernel would make the "right" decision on every access attempt. Here, "right" means consistent with the security policy enforced by the system.

KSOS Final Report

- it would be isolated. The Kernel could not be changed while in operation, and its data bases were also to be protected from modification.

These three principles would allow a system to process information of differing sensitivity. Users whose security clearances were below the System high would be able to share less sensitive data with users having different clearances.

The simultaneous realization of all three principles was to be based on the use of mathematical techniques. The behavior of the system would be mathematically specified and analyzed for compliance with the principles. It was expected that the mathematical rigor would assure the ability to make statements about the System's actions in all possible situations.

Construction of a system which followed these principles required two classes of preliminary work. One class of such work was to develop mathematically precise characterizations of the security policy to be enforced by the system. A second class of work was the development of concepts and mathematical tools by which a system could be constructed and analyzed for compliance with the principles. Specifically, tools and techniques for formal verification of programs were required.

3.3 The Security Policy Model

In 1974, researchers at the MITRE Corporation working closely with their sponsors at the Electronic Systems Division of the Air Force Systems Command (ESD), proposed a model for security that has been used in most subsequent secure system efforts. Briefly, the model consisted of a characterization of a system as a state machine whose responses to external stimuli were determined both by the stimulus applied, and by previous stimuli which had altered the state of the system. By imposing restrictions on the state transitions, one could prove that a system which started in a secure state would remain secure. Here "secure" means no disclosure of information in violation of the security policy.

The model of security policy was based on the mathematical concept of a "lattice". A lattice is a generalization of the notion of the common idea of "less than" or "greater than". Ignoring some mathematical detail, a lattice consists of a set and a partial ordering relationship between members of the set. A partial ordering is a relationship like "less than/greater than" which has been generalized to permit the possibility that two members of the set are not comparable at all.

In the security model proposed by MITRE (also called the Bell/LaPadula Model after the authors of the MITRE reports detailing it), each protected object (e.g., data file, program, etc.) and each active agent (e.g., process or task) is assigned a security level consisting of a hierarchically ordered security classification (e.g. TOP SECRET, SECRET CONFIDENTIAL or UNCLASSIFIED) and a (possible null) set of security compartments. The compartments are typically need-to-know categories such as NO FOREIGN DISSEMINATION or particular project access. Two security levels are equal ("=") if the classifications are equal and the compartment sets are equal. Two security levels, A and B, are related by the partial ordering "<" if:

- security classification of A is below that of B and the compartment set of A is a subset of the compartment set of B. (By the definition of subset the two compartment sets could be equal.)
- the classification of A is equal to that of B and the compartment set of A is a *proper* subset of that for B.

The MITRE model has two basic rules for permissible state transitions:

- (Simple Security Property): An object of security level A may be read by a process of security level B if and only if $A \leq B$, under the above definitions of $<$ and $=$.
- (Security *-property.): (Read as "security star-property") A process of security level B may write information to an object of security level A if and only if $B \leq A$.

This model corresponds well to the operation of the current security system with respect to documents. The simple security property represents the rule which allows a person to read a document if his clearances are at or above the level of the document, and his set of need-to-know categories includes all of the categories of the document. The analog for the *-property is somewhat less intuitive. It is an attempt to prevent processes from downgrading the classification of an object by copying it into an object of lower

KSOS Final Report

classification.

The MITRE model asserts that if the system starts in a state with no violations of these two principles and every action by every process is permitted to complete only if it does not violate the principles, then the system will continue to be secure.

3.4 Mathematical Techniques Required

To describe what the system does in response to actions by processes also required new technology. To analyze whether the security properties are violated by a given action, the details of effects on the system by every action must be stated. The techniques proposed by Parnas for the characterization of system behavior were seen as a feasible way to describe the system. These techniques allowed the analysis of the design independent of the details of its implementation. In a second step, the implementation could be shown to conform to the design using the techniques proposed by Floyd and Hoare.

Considerable uncertainty surrounded the feasibility of these techniques for use in actual systems. To address this question, several experiments were begun.

3.5 Early Experiments

The first of the experiments was begun at MITRE under sponsorship of ESD. This project was intended to build a security kernel for the Digital Equipment Corporation's PDP-11/45 minicomputer. The primary purpose of the project was to demonstrate the viability of the concepts and techniques presented above. (Most of these were actually developed in the early going of what was to become the MITRE 11/45 Kernel project.)

The MITRE project produced formal specifications for the Kernel using the ideas proposed by Parnas. These specifications were then shown to comply with the security model by an exhaustive manual process. The Kernel was then carefully implemented in an attempt to minimize flaws in implementation. The technology for program verification would not permit anything more than demonstrations of program to specification correspondence proofs.

The MITRE project met with good success. Although numerous small problems were encountered, the basic ideas appeared to be sufficiently valid to move on to the next stage. Several projects were initiated by various agencies at a variety of institutions. The ones of greatest relevance to KSOS were:

- development of a secure version of the UNIX operating system at UCLA under DARPA sponsorship.
- development of a secure UNIX at MITRE under ESD sponsorship.
- research on techniques for systems development at SRI International under sponsorship of several agencies.

3.6 KSOS Beginning

By the middle 1970's, the early projects were going well enough to suggest the feasibility of developing a "production quality" secure computer system. The system would be based on UNIX to get maximum benefit from the large and growing UNIX application base. To minimize the risks of the project, the system was to be based on the DEC PDP-11/70. Newer minicomputers were known to be in the offing, but staying with the 11/70 avoided the additional complexity of a new computer system.

In late 1975 a notice appeared in the Commerce Business Daily inviting interested contractors to submit their qualifications for the development of a "Secure Minicomputer Operating System". In the spring of 1976 a formal competitive RFP was issued for the project to be accomplished in two phases. More than one contractor would be selected for the first phase. The intent was to select the best overall design for a subsequent implementation phase.

Two design contracts were awarded, one to TRW Defense and Space Systems Group, and one to Ford Aerospace & Communications Corporation (FACC). Each contract was for an eight month design phase which included production of System Specifications (also called A Specifications), Computer Program Development Specifications (B Specifications), and several other plans for how the system would be developed and shown to comply with the desired security properties. The FACC team included SRI International as a subcontractor for verification and other formal mathematical tasks. In April, 1977 the

KSOS Final Report

Government selected the FACC team to continue into implementation.

At approximately the same time, the name Kernelized Secure Operating System (KSOS) was chosen for the project due to trademark issues on the use of the name UNIX.

KSOS Final Report

4. GOALS, OBJECTIVES, and REQUIREMENTS

In this section, the term "goals" is used to describe the desired overall effect of this project and similar projects within the computer science community. "Objectives" relate to how much closer to the goals this project was intended to carry the entire community. "Requirements" are derived from goals and objectives. They describe both the constraints and expected results of the project itself.

4.1 Goal

The primary goal was a secure system which would address the need described previously. This goal will have been achieved when/if the appropriate security authorities within the DoD community are finally able to certify a single system safe to be utilized simultaneously by users constrained to different security classifications or compartments. The system in question could be a general purpose time-sharing system, a network, or a common database containing compartmented information.

4.2 Objectives

The KSOS program addressed a number of objectives which are considered stepping stones to the aforementioned goal. These objectives relate to the structure of the system, the development methodology, and the verification that the desired security properties have been successfully achieved through the use of the methodology and system structure. An additional objective was that the system be directly usable in a manner consistent with the overall goal. The top level objectives were:

Structure— The structural objective was a kernelized system, that is, one decomposed in such a way that all of the security related features could be placed in a small component of the system, thus reducing the size and complexity of code implementing security.

Formal security properties— The security definition objective was to test out the best available mathematical model of the DoD security policy to see if this mathematical representation was an accurate interpretation of the existing DoD security policy.

Development methodology— The methodology objective was to see whether the formal development methodology could yet be effectively employed on a project as complex as an operating system.

Verification— The verification objective was to determine whether the technology was (or could be) sufficiently advanced to provide mathematical proofs of system properties and behavior, thereby eliminating the possibility of having overlooked some nuance in the system which could later be exploited.

Usability— The usability objective was that a system structured, developed, and verified to have the required security properties be also usable in the environment for which it was intended.

4.3 Requirements

The statement of goals and objectives given above represent a retrospective view of what the project members believe must have been in mind when the government initiated this effort with the RFP. From that point on, all of the goals and objectives are reflected in a set of precise requirements placed upon the system to be met by the developers of the system. These precise requirements are stated in the Request for Proposal, and are restated in the System Specification. The requirements are intended to be testable and objective measures. For a complete treatment of the requirements, the reader is referred to the RFP and the A-Specification.

For the purposes of this document we will simply summarize the requirements in a way which will relate them to the objectives stated above.

4.3.1 Structural Requirements

The system is required to be structured as a kernelized system, decomposed in such a way that all accesses from one process to another process would be mediated by the security kernel. The primary requirements on the kernel are:

- It will be complete. Every access attempt will be mediated by the kernel. Some of these checks will be made by hardware, others by software.

KSOS Final Report

- It will be correct. The kernel will make the correct decision on every access attempt. Correct, in this case, means "consistent with the security policy enforced by the system".
- It will be isolated. The kernel cannot be changed while it is in operation and its databases cannot be modified by unauthorized user programs.
- It will be general purpose. The kernel will be amenable to support of other applications or systems. The kernel is expected to be used as a basis for other systems besides UNIX, particularly for applications systems. The basic kernel design is to be usable on another minicomputer system to support applications.

4.3.2 Formal Security Requirements

The requirement for formal security properties was that the kernel be provably secure, based on a security kernel approach and formal specification and verification of the kernel itself. The security policy of the system was to be built upon the Bell and Lapadula model.

4.3.3 Development Methodology Requirements

The development methodology to be used was the Hierarchical Development Methodology (HDM) as developed at SRI.

4.3.4 Verification Requirements

The verification requirement was that the formal specifications of the system design be verified against the mathematical security model using the theorem prover developed by Boyer and Moore at SRI.

4.3.5 Usability Requirements

The usability objective was that the system be based on the PDP 11/70 hardware, that it present to the user a UNIX operating system, and that this operating system would perform no less than half as efficiently as the UNIX operating system without the security properties.

Additional usability requirements included the administrative support for a community of users on a single system.

KSOS Final Report

5. DEVELOPMENT

KSOS is decomposed into three major components: Kernel, Emulator, and Non-Kernel Security Related software (NKSr). The development effort, however, had four major components. The network software, which was to consist partially of additional functionality within the Emulator and partially of NKSr software, had a separate design and development team. This section deals with each of these design and development efforts and their interrelation.

5.1 The Kernel

5.1.1 Introduction

To make the internal interfaces more simple, the KSOS Kernel can be viewed as being composed of three layers:

- a primitive multi-programming system (including a primitive synchronization mechanism) which runs processes until they voluntarily relinquish control.
- a simplified operating system providing primitive I/O, process management and segmentation management.
- an interface layer composed of virtualized system call interfaces for user process management, segmentation management, and both synchronous and asynchronous I/O to devices, terminals, extents, and files.

All objects supported by the KSOS Kernel have an associated Secure Entity Identifier (SEID). The basic objects supported by the Kernel are:

- files: a linear array of blocks of data.
- devices: any "unmounted" peripheral device.
- terminals: to which secure and non-secure paths exist.
- segments: contiguous portions of physical memory or swap space.
- processes: the active agents in the KSOS system.

5.1.2 Kernel I/O

5.1.2.1 Features

The KSOS Kernel I/O is patterned after the UNIX I/O paradigm. In particular, to perform I/O, a process must first "open" the I/O object. Once the object has been opened, the process can perform read and/or write operations as specified in the "open". When the process has completed its usage of the object, it will then "close" it. The advantage of using this paradigm in the KSOS Kernel is that security flow checks to and from objects can be performed at "open" time and thereafter ignored. Since a process cannot perform I/O without having successfully performed "opens", then the security of the system is maintained.

Unlike UNIX however, the KSOS Kernel I/O does not support caching of file blocks. Instead, file I/O is done directly to and from a process' virtual memory (as in DEC's RSX-11 operating system.) The reason that the KSOS Kernel does not support file block caching is that it was assumed that buffer cache management would be too complicated for inclusion in a secure, verifiable operating system. In hindsight, this assumption may not have been valid.

Another important distinction of the KSOS Kernel I/O with respect to UNIX is the robustness of the filesystem structure. Due to careful designing, the only damage made to a KSOS filesystem due to a power fail or unexpected halt is the loss of a few blocks of free space. This aspect of the file system can be contrasted with UNIX, where such conditions can mortally wound a filesystem. However, such damage prevention is attained only by accepting certain performance penalties.

In addition, the KSOS Kernel provides some unique features not found in other comparable operating systems. Included are:

- An exclusive-use "open" mode is provided to guarantee access to an object by exactly one process at a time. When a process has an object open via "exclusive-use", then it is guaranteed that no other process in the system has the same object open. Unfortunately, since exclusive-use is pervasive, it is

KSOS Final Report

possible to use it as a timing channel. The bandwidth of the exclusive-use timing channel is limited by adequate restrictions on its usage.

- All terminals have several paths over which I/O may be done. In effect, each path to a multiple security level terminal acts as a single security level virtual terminal. In this way, critical NKSR processes will have "secure" paths to the terminal, and hence, to the user. As part of the terminal path management, a "break" character is recognized by the Kernel as a special attention character which causes a switch to path number zero. By convention, the user's Secure Server (see NKSR) is attached exclusively to path number zero.
- The subtype mechanism is provided to allow construction of special files which ordinary processes are not allowed to access (particularly for write access.) Subtypes were perceived to be the only means by which the NKSR is capable of protecting the integrity of, among other things, the UNIX directories. However, proper use of integrity compartments (not currently implemented in KSOS) could probably have been used, perhaps more efficiently, for the same purpose. In addition, subtypes can be used to construct "extended types" whose management is the sole domain of an "extended type manager".
- Asynchronous I/O is supported by the KSOS Kernel. To perform asynchronous I/O, a process needs to perform a regular I/O call with the asynchronous flag set. In this case, the Kernel I/O system will set up the I/O and immediately return to the caller. When the I/O completes, the Kernel will then send an "I/O completion pseudo interrupt" to signify the completion of asynchronous I/O.

5.1.2.2 Issues

The Kernel I/O, like all of the other portions of the Kernel, has several issues surrounding its design. Of notable importance are the issues of subtypes, extents, asynchronous I/O, terminal paths, and terminal I/O.

5.1.2.2.1 Subtypes

Subtypes, as described above, are a means for guaranteeing the management of particular extended types. In KSOS, the only use for this functionality is in the construction of UNIX directories out of the Kernel's flat file namespace. The issue here is the advisability of the inclusion of subtypes in the Kernel design. In particular, are subtypes necessary for the implementation of probable Kernel applications? The answer to this question is only clear if one has a clear idea of the probable applications that the Kernel may be required to support. It is likely, however, that subtypes would provide an excellent mechanism for implementation of a wide range of applications, particularly those with no intrinsic security implications.

Although potentially very useful, subtypes do cause a noticeable increase in the complexity of the Kernel. Since complexity often is translated into implementation problems and performance difficulties, great care should be taken before including subtypes into the design of any operating system.

5.1.2.2.2 Extents

A KSOS extent is a contiguous area of an addressable device (viz., disks or DECtape), which may be used by a single user as one large file of fixed size, or which may hold a file system permitting many variable length files.

The extent mechanism, not unlike the IBM notion of "mini-disk", is supported by the KSOS Kernel I/O to allow rational management of the KSOS disks. The UNIX operating system provides the same functionality by the use of "special files" which access unmounted "mini-disks" as if they were devices. Like subtypes, extents are an elegant and useful concept. Also like subtypes, the inclusion of extents into the Kernel I/O cause an increase of complexity; and hence, the inclusion of extents should be very closely scrutinized.

In spite of the added complexity, however, rational management of unmounted file space (that is a "mini-disk" or extent) is very important in a secure (or reliable) operating system. Furthermore, the inclusion of extents helps to remove any ambiguities of filesystem management. (For example, in the UNIX operating system, a filesystem may be mounted onto a directory or a file. In the KSOS Kernel, they can only be mounted onto an "extent".)

KSOS Final Report

5.1.2.2.3 Asynchronous I/O

One of the more controversial issues of the Kernel design is the concept of asynchronous I/O. The primary areas of disagreement involved the two questions:

1. Should asynchronous I/O be included in the KSOS Kernel?
2. If so, what flavor of asynchronous I/O should it be?

Whether or not asynchronous I/O should be included was not resolved until just before implementation began. The two strongest arguments in the debate were:

- Pro: If done properly, the inclusion of asynchronous I/O will not significantly impact the design and implementation of the Kernel I/O system.
- Con: Asynchronous I/O is an unnecessary aberration which will cause untold problems in design, implementation, and eventually performance.

Unfortunately, both arguments are correct.

Once it was decided that asynchronous I/O was wanted and desirable, the exact flavor of the asynchronous I/O needed to be decided. Since the design of the Kernel I/O system had progressed too far to warrant a re-design, asynchronous I/O was added on in the form most congenial to the rest of the Kernel I/O system. Hence, only one asynchronous I/O at a time is allowed per device and when attempting to perform asynchronous I/O, a process must wait until the device becomes available before it is released by the Kernel.

5.1.2.2.4 Terminal Paths

As discussed briefly above, some mechanism was necessary to guarantee a "secure" path between the user at a terminal and a "trusted" process on the system. (This functionality is necessary to prevent "spoofing.")

The design of the Kernel I/O system accommodated this problem by the inclusion of the terminal path mechanism. By definition, every terminal would have more than one path to it, each of which would represent a virtual terminal. Also by definition, each virtual terminal would operate at one, and only one, security level at a time. This design is an elegant way of providing the functionality needed for the "secure" or "trusted" path. Unfortunately, like so many other features of the KSOS Kernel, it added complexity to an already too complex operating system.

5.1.2.2.5 Terminal I/O

Originally, terminal I/O was thought of as a very simple mechanism in the KSOS Kernel. In theory, most of the terminal support would come from processes outside of the Kernel. As in many such cases, the original analysis of the KSOS Kernel failed to discern the performance penalties associated with placing terminal support outside of the Kernel. Hence, when this did become evident, more and more of the terminal support was pushed back into the Kernel. Unfortunately, the resulting terminal interface is awkward to use and does not support the correct level of virtualization.

5.1.2.3 Summary

The KSOS Kernel I/O plays a major part in the design and implementation of the KSOS Kernel. As in all comparable systems, decisions in the I/O area greatly affect the design of the rest of the operating system and ultimately reflect the performance possibilities of the system in general. The KSOS Kernel I/O meets or exceeds all reasonable requirements (except performance) placed upon it by the original procurement.

5.1.3 Kernel Process Management

5.1.3.1 Features

The Kernel Process Management (KPR) resides at two levels. The low level provides primitive (nucleus) process management in form of Modula language process support. The high level provides user process management. User processes are the processes visible at the Kernel interface. All of the processes in the system are either Kernel processes (nucleus processes) or user mapped processes (nucleus processes).

KSOS Final Report

instantiating a user process.) Kernel processes always execute in the Kernel and never anywhere else. User mapped processes will execute inside and outside of the Kernel.

Only five functions are supported for nucleus processes. They are:

1. Nucleus (or Modula) process creation.
2. Nucleus (or Modula) signal wait which suspends execution of the calling process.
3. Nucleus (or Modula) signal send which immediately starts the process which is first in the signal wait queue; temporarily suspending the calling process. (Note: that a "device" process is not suspended on a signal send and the signal receiving process is not immediately resumed.)
4. Nucleus (or Modula) signal awaited, a boolean valued function which returns true if, and only if, a process is waiting for the given signal. The signal awaited call never suspends execution of the calling process.
5. Nucleus (or Modula) process deletion.

The upper limit to the number of nucleus processes declared in the system is static and is a compile-time constant. At this primitive level, the only process scheduling provided is the Modula signal send, wait, and awaited functions. This fact has significant implications with respect to the performance of the system.

User processes are supported in a much richer way. Like all other Kernel exported objects, user processes have a SEID, they are assigned a level, and their status can be read and written like any other object. In addition, the following types of functionality are supported:

- Process creation and deletion. Process creation is supported by three distinct Kernel calls:
 1. `K_fork` which performs a UNIX-like process "fork" where a process is transformed into two identical processes (the calling process known as the parent.)
 2. `K_invoke` which performs a UNIX-like process "exec" where a process's virtual address is manipulated.
 3. `K_spawn` which is logically a `K_fork` followed immediately by a `K_invoke`.
- Interprocess communication (IPC). The KSOS Kernel IPC mechanism is a transaction based system where every process has an IPC queue in which 16 byte messages can be stored. Since the IPC mechanism is directed by the SEID of the receiving process, certain security and integrity implications must be thought out.
- Pseudo interrupts. The KSOS Kernel pseudo interrupt mechanism imitates the interrupt mechanism of the PDP 11/70 processor. In particular, every process's supervisor domain data space is assumed to have six interrupt vectors in it. An interrupt vector contains space for the interrupted program counter, the interrupted processor status word, the interrupted pseudo interrupt level (analogous to the CPU interrupt level), the IPC formatted pseudo interrupt message, the new program counter, and the new processor status word (these last two are analogous to the pc/ps pair placed in the hardware interrupt vectors.)

Pseudo interrupts are used by the IPC mechanism, the asynchronous I/O mechanism, the fault detection and correction mechanism (e.g., an illegal instruction pseudo interrupt), and by the timing alarm mechanism.

Since manipulating a user process' stack was perceived as undesirable, the pseudo interrupt mechanism simply changes a process' program counter and processor status (retrieved from the pseudo interrupt vector) and stores the appropriate information into the pseudo interrupt vector, and changes the current pseudo interrupt level of the process. It is the responsibility of the user process to save and restore registers and any other process state as is needed.

- Process scheduling. The user process scheduling is based upon a wait/ready paradigm in which all processes are in one of two states at all times: waiting or ready. When ready, a process can be run simply by performing a process context switch; i.e., all requested system resources have been assigned to the process. Due to the different types of resources, there are three different stages of "waiting". They are:

KSOS Final Report

1. Waiting for user level events (e.g., waiting for an IPC message, waiting for an timer alarm, etc.)
2. Waiting for the swapper to swap in the process.
3. Waiting for the nucleus process mapper to instantiate the user process with a nucleus process.

5.1.3.2 Issues

Several deep lying issues exist in the Kernel Process Management (KPR). Foremost among these are process scheduling, process swapping, interprocess communication and synchronization, process pseudo-interrupts, and process bootstrapping.

5.1.3.2.1 Process Scheduling

One of the most profound problems facing the KSOS system is the question of user process scheduling. In a timeshared secure operating system, the question of the relative priority of competing processes is very difficult. Assigning priorities must be made upon some very well defined lines: for example, an NKSR process should receive better response (on the average) than some arbitrary user process; and a low level user process should not be able to degrade system performance enough to disturb a high level user process.

The user process scheduling issue, if solved at too low a level in the operating system, can cause severe difficulties in design and implementation as well as in overall system performance. In particular, the KSOS Kernel user process scheduling mechanism relies heavily upon the priorities assigned to user processes. In this way, policies which are formulated outside of the Kernel can be enforced by the Kernel.

Lastly, overall system performance (as affected by process scheduling) is designed to degrade gracefully and is optimized for a medium system load. For this reason, when comparing Kernel performance with other operating systems, it will probably be best to compare medium system loads instead of minimum system loads.

5.1.3.2.2 Interprocess Communication and Synchronization

Interprocess communication (IPC) and interprocess synchronization (IPS) are two of the most important aspects of an general purpose operating system design. The IPC and IPS mechanisms provided by the KSOS Kernel were designed to be general, non context-sensitive (for example, independent of "process families") mechanisms. In particular, the KSOS Kernel IPC and IPS mechanisms were combined into one.

The actual packaging of the IPC and IPS mechanisms was a hotly debated issue which never satisfactorily concluded with a technologically superior design. However, two aspects of the IPC and IPS mechanism (as implemented in the KSOS Kernel) need to be mentioned:

- The mechanism does not provide a high enough bandwidth for effective IPC usage.
- The mechanism does not provide suitably encapsulated IPS for efficient usage.

Thus, in retrospect, the current KSOS Kernel IPC/IPS mechanism is inadequate on two fronts.

5.1.3.2.3 Pseudo Interrupts

The issue of interrupting processes runs very deeply into the heart of the design of the KSOS Kernel applications. Several approaches were evaluated and rejected in favor of the pseudo interrupt mechanism. In particular, it was believed that the most important Kernel applications would be operating systems emulators which would be "operating system-like" themselves. Hence, an interrupt mechanism not unlike those used by processors was considered a likely choice. In this way, the current pseudo interrupt mechanism (based on the PDP 11/70 interrupt mechanism) was born.

Several problems manifested themselves during the design and implementation of KSOS which were directly related to pseudo interrupts. The most important among these problems dealt directly with the usage of pseudo interrupt levels and the inability to change it without making a Kernel call. Whereas changing the PDP 11/70 processor interrupt level is very simple and fast (hardware supported), changing a pseudo interrupt level is relatively slow and complicated.

KSOS Final Report

5.1.3.2.4 Process Bootstrapping

Early in the design of KSOS, it was decided that process bootstrapping (i.e., the initialization of processes from machine language objects) would be performed as one of the functions of the NKS. Furthermore, the original Kernel interface design was inadequate to properly perform this task. Hence, improvements and support had to be added to the Kernel to allow the NKS to perform the process bootstrapping. In all, it would have probably been simpler and more efficient to have done the bootstrapping inside of the Kernel.

5.1.3.3 Summary

The fundamental issues of the Kernel Process Management surround the objectives of the system. The loosely coupled nature of the IPC mechanism, for example, was unnecessary for the construction of the UNIX emulator. The concepts of general purpose and secure made the process management ungainly and inelegant.

5.1.4 Kernel Segment Management

5.1.4.1 Features

The Kernel Segment Management (KSM) is the part of the Kernel which supports process swapping and segment manipulation. The KSM provides segmentation features not found in the UNIX operating system. Like UNIX, however, the KSOS Kernel provides a process swapping environment where no process is allowed to execute without having its entire virtual address space (its "mapped in" segments) swapped in.

As stated above, the KSOS Kernel provides a much richer notion of segments (and of virtual memory) than does the UNIX operating system. In particular, a process may have up to some predetermined number of segments (sixteen in the standard release), some of which may be "mapped-out" (not currently part of the virtual address space), some of which may be sharable (with other processes), which may reside in either supervisor or user domain of the PDP 11/70, and which may reside in the text (instruction) or data spaces of the PDP 11/70. Furthermore, each of the mapped-in segments is separately swappable.

5.1.4.2 Issues

The issues of the Kernel Segmentation Management were tame in comparison to the rest of the system due to a common regard for the proposed design. The most important issues concerned the number of allowed segments per process, and the usage of segments (especially shared segments).

5.1.4.2.1 Segments Per Process

Early in the design phase, it was felt that 16 segments per process would be sufficient to implement likely KSOS Kernel applications. However, during the development, 16 segments seemed like too many to the Kernel designers and too few for the applications designers (especially the UNIX Emulator).

The problem stems from the fact that all of the segments of a process are separately swappable. Therefore, when swapping a process into memory, each segment has to be swapped independently from the others. This separate swapping causes a substantial performance penalty. Furthermore, if a 50 process maximum is employed, then for each process to have a full complement of segments, $16 \times 50 = 800$ segment table entries will be necessary. Currently the KSOS Kernel will only support 256 segments (5.1 segments/process average) because of the KSOS Kernel address space limitations (40 Kbytes maximum static data addressing.)

On the other hand, applications designers felt that 16 segments was not enough. For example, for the UNIX Emulator to function properly, every user process would require (at a minimum) 6 segments (one text, one data, and one stack segment for each of the supervisor and user domains). For just a minimum configuration, the average number of segments allowable per process (5.1) is exceeded.

Hence, a paradox was introduced: the Kernel cannot support more segments, the applications need more. The only known solution to this problem will require extensive redesigning of most of KSOS, especially the Kernel and UNIX Emulator.

5.1.4.2.2 Segment Usage

Segments in KSOS, unlike UNIX, are separately accessed objects (accessed via SEIDs.) To provide this access requires considerable complexity not found in operating systems like UNIX. In the KSOS Kernel, a

KSOS Final Report

lot of source code could easily be eliminated if segments became more UNIX-like. However, to do so would unalterably change the "flavor" of KSOS.

For example, the KSOS Kernel concept of shared segments is very desirable to implement high-bandwidth interprocess communication. In this case, the sharing processes could both read and write the same segment if they so desired; a not necessarily desirable property. The most disappointing aspect of shared segment IPC is the lack of a good interprocess synchronization mechanism. A binary semaphore mechanism to synchronize on shared segments was proposed too late to impact the design of the Kernel Segment Management or the Kernel Process Management.

5.1.4.3 Summary

The Kernel Segment Management seems to the uninitiated observer to be the simplest of the major components of the KSOS Kernel. However, on careful examination, its design and Kernel interface functionality carry a great impact on the design of the other parts of the Kernel and of all applications. It was in the Kernel Segmentation Management that the greatest opportunity for significant improvement over contemporary operating systems existed. The ability to have separately swappable and accessible segments is a major step forward; but too inefficient for the KSOS environment.

5.1.5 Conclusion

The Kernel design and implementation, though confronted with many difficult issues and problems, managed to meet every major design goal except performance. In this regard, it is important to realize that no major performance tuning and optimization has yet been performed. In comparison with other similar operating systems, it is not unreasonable to expect substantial improvements in performance as a result of run-time experience, performance tuning and optimization, and selective redesign and re-implementation.

5.2 Emulator

5.2.1 Introduction

The UNIX Emulator has a well defined basic function: to create the objects of the UNIX interface from the more primitive objects provided by the Kernel interface, and to synchronize the use of these UNIX objects among members of the "process family". (In KSOS, a UNIX process family is all of the processes associated with a given user login, on a given terminal, at a given security level.) In addition, the Emulator was to contain much of the support for the computer network interface (described in a separate section).

In the early stages of the design of KSOS (actually until after the PDR, when the detailed design began) the Emulator was viewed simply as a set of subroutines existing in Supervisor domain of the 11/70, which were called by TRAP instructions. It was planned that a small amount of state information would be saved in tables maintained by the Emulator and that all communication necessary between processes would be done using the IPC mechanism provided by the Kernel.

It was discovered, however, that to faithfully emulate the subtle process and I/O semantics of UNIX that potentially more extensive sharing of information was necessary among processes. At the UNIX user call interface there are several subtle side-effects which lead to indirect interaction between processes in the same process family. Some of the status information may be altered by any member of the family, with the effects of such alterations being visible to all of the members of the family. Deviation from these subtle semantics of the UNIX user interface would pose substantial conversion problems for software to be moved between UNIX and KSOS, and one of the major requirements of the system would not be satisfied.

Thus, the Emulator needed to provide a means for controlling resources which would be shared among the processes of a "process family" (all the descendants of a single user login at a given level). Some of these shared resources were used only within the Emulator for performance enhancement (such as the buffer cache), and others because they were needed to emulate user visible resources or side-effects (such as sharing of open files inherited from a fork).

By the October 1978 IPR, the Emulator had emerged as an operating system in itself, running on a "Kernel Machine", creating UNIX level resources from the resources provided by the Kernel, and synchronizing and controlling their use by members of the process family. This functionality is precisely what is done by any operating system.

KSOS Final Report

The difference between the Emulator and other operating systems lies in two areas. First, the underlying machine upon which the Emulator runs is considerably more sophisticated than typical real hardware bases. The Kernel creates many objects (files and devices, segments, processes, and subtypes) that have a fairly complex hardware realization. Thus, the Emulator is able to create the objects of the UNIX user interface out of more powerful underlying objects. The second difference is that the Emulator is a distributed operating system. Logically, there is a distinct Emulator running in each member of the process family. These separate Emulators can communicate over the Kernel IPC facility or via shared data segments. They need to manage shared segments which hold and communicate process family data, port data, and network data. Coordination of the use of the shared segments could be done with IPC messages.

Complicating the Emulator's task is the fact that at any time, the Kernel may elect to switch to another process. The Emulator does not have at its disposal a convenient means to lock out other members of the process family while modifying complex shared structures. There is no way provided by the Kernel to guarantee long (tens to hundreds of instruction executions) periods in which a given Emulator will not be preempted, and the CPU given to another process, possibly one in the same process family. All that can be guaranteed is that individual instructions operate atomically.

Further design analysis revealed that other areas which would require extensive interprocess coordination and information sharing in the Emulator were the network and terminal handling. Both of these required special treatment in the Emulator. For example, because of Kernel requirements, the Emulator had to guarantee that, at any instant, exactly one process in the process family (the Master in the current design) had a read to the terminal outstanding. Mastership would pass from process to process in the family, following terminal I/O activity.

Alternatives to the Master concept which were considered included a "Family Process" which would serve as the I/O daemon for the entire process family.

As a result of these considerations, the Emulator was designed around a Runtime Executive, which provided an integrated set of general mechanisms for synchronization and multiple virtual IPC channels. Also, a set of linguistic constructs, implemented as macros, provide a discipline for the use of shared data. The Runtime Executive was designed to be sufficiently general so that it could be used not only by the Emulator, but also by other programs (such as NKS or applications) which were envisioned to run directly on the Kernel.

The Emulator is said to have a "top half" and a "bottom half". The top half is the synchronous part of the operating system which interprets UNIX system calls made by the program in the user domain of the process. The bottom half is the asynchronous portion, processing pseudo-interrupts and performing certain Kernel interactions. Event channel handlers, similar to device drivers in UNIX, handle the interaction between the top and bottom halves.

The Emulator is divided into 5 major pieces:

- Runtime Executive,
- I/O System,
- Process System,
- Miscellaneous Services,
- TCP Services,

which are discussed in more detail below.

5.2.2 Runtime Executive

The Runtime Executive is a collection of functions identified as being common to subsystems built directly upon the KSOS Kernel, and is intended to provide a solution to several problems and complexities engendered by the distinct environment provided to processes running on the KSOS Kernel. The Runtime Executive consists of several modules: Kernel Call Library, Directory Operations, Synchronization Primitives, and Event Manager.

KSOS Final Report

5.2.2.1 Kernel Call Library

This is a set of small direct (machine) code sequences, callable from the implementation language, which formats the parameters and generates the EMT instruction for the various Kernel calls.

5.2.2.2 Directory Operations

Actually two modules: Pathname interpretation and Directory Manager Interface. Since all normal directories are created with the subtype read permission granted to all, an unprivileged Emulator process can read directories (discretionary access modes permitting) without intervention on the part of the (NKS) Directory Manager process. Thus, most pathname interpretation can be done by the Emulator itself. For the maintenance of directory contents, or the reading of search-only directories (ala UNIX 111 mode), the Emulator must use the Directory Manager. This piece of NKS software operates as a separate process and has the discretionary access domain of the directory subtype owner, and hence, the privilege to manipulate the directory subtype. The Directory Manager Interface module in the Emulator hides the details of spawning the Directory Manager process, and the communication of parameters and results between the processes.

5.2.2.3 Synchronization Primitives

These primitives provide the appropriate abstractions and disciplines necessary to minimize the likelihood of programming error in the concurrent programming environment of the Emulator. The Synchronization Primitives accommodate arbitrary sets of (related and unrelated) processes accessing any number of shared variables on various shared data segments. The discipline is decomposed into several hierarchically ordered levels of abstraction:

- **Monitor Procedures** may be constructed as a programming discipline by the use of the shared variables in a structured way. A set of data shared among several processes is identified and grouped together with the procedures which represent the valid operators on that data. In order to guarantee integrity of the shared data, the monitor procedures use the shared variable discipline.
- **Shared Variables** represent a discipline and mechanism for concurrent access to variables available to several processes. The logical consistency of such variables are guaranteed by embedded control information manipulated by the lower level mechanisms.
- **Unconditional and Conditional Critical Regions** are used to guarantee the indivisibility of any reference to or manipulation of a shared variable. The execution of sections of code which access a particular shared variable are constrained to mutually exclude each other in time, and are known as Critical Regions with respect to that variable. Critical Regions are represented by linguistic constructs, implemented by macros, which enclose the critical code.
- **Semaphores and Event Queues** are used in a well defined and structured way (packaged in a linguistic construct) to implement Unconditional and Conditional Critical Regions. Each Shared Variable has associated with it a mutual exclusion semaphore. An Unconditional Critical Region with respect to a Shared Variable is, therefore, a Wait (or P) operation, followed by the body of the critical section, followed by a Signal (or V) operation. The Semaphore provides a first-in-first-out waiting queue for processes waiting to enter their Critical Regions. Conditional Critical Regions are constructed by providing an additional waiting queue associated with the Shared Variable. A process executing within a Critical Region is able to execute a primitive requesting to be placed on this queue and be suspended until an arbitrary boolean condition holds. Event Queues are available to permit more explicit scheduling of processes waiting for particular events or classes of events.
- **Multiprocessor Synchronization and Kernel IPC** are the low level mechanisms used to make Semaphore and Event Queue operations work. Since semaphores are, themselves, shared variables and the primitives which manipulate them are subject to the same undesirable time dependent interactions that they were introduced to prevent at the higher abstract levels. Thus, this mechanism provides primitives which are called by Semaphore and Event Queue operations so that they might behave as monitor procedures with respect to the semaphores. The synchronization performed takes into account the (virtual) multiple processor environment in which which processes running on the Kernel are operating.

KSOS Final Report

- Indivisibility of load, store, increment and decrement operations on shared hidden control variables is the ultimate foundation of mutual exclusion.

5.2.2.4 Event Manager

This module encapsulates the interprocess communication (IPC) facility provided by the Kernel and provides multiple virtual interprocess communication channels and a hospitable environment for the construction and execution of event driven code within the supervisor domain of a process. It contains low level asynchronous IPC dispatching and IPC transmission functions, and a handler for timeouts used by the Emulator. It also provides buffering for asynchronously received IPC's which are to be handled synchronously.

5.2.3 I/O System

Most of the functions and structures of the Emulator I/O system are shared by processes within the process family. At the top level, I/O requests are decoded to determine the type of I/O object so that the request may be dispatched to the appropriate module within the I/O system. The various modules of the I/O system are: Files, Pipes and Ports, Terminals, Devices, and the Buffer Cache.

5.2.3.1 Files

The basic dispatching for the I/O system is done in this module. I/O operations may be directed to files, devices, terminals, pipes and ports, or network connections. The System Call Interpreter dispatches generic I/O related system calls to this module, which determines the object type and calls the proper I/O function.

In addition, all of the operations which pertain strictly to files (e.g., file open, file read, file write), or are type independent (e.g., chmod, stat, etc.), are handled in this module.

5.2.3.2 Pipes and Ports

This module provides the UNIX level interprocess communication. Pipes are a standard UNIX feature; ports however, which provide interprocess communication between unrelated processes, are modeled after the RAND ports. Both pipes and ports are implemented using shared segments.

When a port is created, a directory entry is created for the port. Named ports can be opened for writing by processes with write permission, just like an ordinary UNIX file. A writing process need not be aware that it is writing to a port. Only one instance of open for reading is permitted on a port at a time. When creating a port, a mode may be specified which will cause the Emulator to provide header information for every separate write to the port. This is transparent to the writer, but the reader may interpret the header information to identify distinct writes. A mechanism is also provided to allow a process to suspend its execution until either a timeout event has occurred or one of several conditions for waking up has been met. For each open port it is possible to set these wake up conditions: wake up when written, wake up when read, and wake up when empty.

Other user calls exist for discovering the number of bytes available for reading and writing on a pipe or port, and for writing an end of file on a pipe.

5.2.3.3 Terminals

This module contains the functions and data structures needed to handle UNIX terminal I/O. Since the Kernel support for terminal I/O is kept to a minimum, almost all of the buffering and processing for terminals is handled in the Emulator. Such processing includes canonicalization, character mapping, erase and kill processing, delay padding, and control character echoing.

The Terminal Handler is made up of top and bottom halves. The top half operates synchronously with user calls. The bottom half operates asynchronously, driven by event messages such as I/O completion messages and event messages from the top half. The top and bottom communicate through shared data buffers and event messages. Since the Kernel requires that there be a read request to a terminal for input to be accepted and since UNIX compatibility requires asynchrony between UNIX user read requests and character input from terminals, there must always be an asynchronous read operation outstanding from the Emulator. This read operation provides the Kernel with a buffer for the raw character input. Within a process family there is one outstanding read call to the Kernel at any given time for each open terminal. The process doing the current read is called the "read master" for the terminal. If more than one process in the

KSOS Final Report

family has the terminal open and is reading, the read mastership migrates to the process doing the last reading. Data read by the current master is placed in the process family read buffer, from which any process in the family having the terminal open may read.

5.2.3.4 Devices

Devices are defined as objects of Kernel I/O operations which are not files, extents, or terminals. Magnetic tape is a good example of a device. The Device module is responsible for creating the UNIX level device abstractions. The module contains device handlers which manage the peculiarities of particular devices. For example, there is only one Kernel SEID for each tape drive, but in UNIX there are several logical devices for that drive (representing different modes such as density, rewind-on-close, etc.) A tape device handler in the Device module would handle the different modes of a tape drive by altering internal status and issuing Kernel calls to change the device status.

5.2.3.5 Buffer Cache

This module provides the facility which buffers data requested by user read calls and generated by user write calls. A fixed number of buffers are available for this purpose and for other transient needs on the part of the Emulator. In addition to simple buffering, this set of buffers is managed in cache fashion so as to minimize the number of Kernel calls performing block I/O functions.

5.2.4 Process System

This subsystem contains most of the routines which handle UNIX user calls other than those dealing with I/O. In general, it handles all functions dealing with the process's status and its relation to the process family. Some of the process system functions are built directly upon Kernel provided functionality. Others are created and maintained entirely by the Emulator. The process system is responsible for the creation and growth of user domain segments, handling the current working directory, passing of signals, generation of User ID's within the process family, and for making the proper Kernel call(s) in response to user calls to set the process User ID or advisory scheduling priority. The process system also gathers information from all areas to handle status retrieval requests.

5.2.5 Miscellaneous Services

This section of the Emulator, as its name implies, provides a home for a number of services: the System Call Interpreter, Assembly Language Support, and the Emulator Initialization. The System Call Interpreter is the dispatcher for UNIX user system calls. It is responsible for fetching the proper number of parameters from the user domain and dispatching the call to the proper routine within the other Emulator subsystems. The Assembly Language Support contains miscellaneous assembly language routines (except for the Kernel Call Library) which exist either to "utter" instructions which the C compiler will not, or because of an overwhelming requirement for speed. The Emulator Initialization is the top level driver calling the initialization routines in the various modules of the Emulator in order to prepare the Emulator for the running of user mode programs.

5.2.6 TCP Services

This portion of the Emulator contains the user oriented part of the network subsystem, using the Transmission Control Protocol as the end to end protocol. TCP Services provides the interface between the TCP interpreter and the Network Daemon (NKSr). The Emulator Network Control Program consists of a top half and a bottom half. The top half operating synchronously driven by user network I/O calls. Any user oriented processing, such as formatting of segments, occurs here. The bottom half consists of various event handlers controlling the network events received from the Network Daemon, a connection timeout event handler, and a retransmission timeout event handler. It performs segment processing including validation of the segment, processing of control information, queueing the segment for the top half, and sending an event notice to the top half for notification.

5.2.7 Testing and Integration

The Emulator implementation effort produced several tools and aids to the informal testing process. Since the Emulator is a concurrent program, every effort was made to thoroughly test components in a bottom-up fashion using unit test frames, so as to reduce the costly process of debugging. This approach was even more successful than expected: considering the size and complexity of the Emulator, surprisingly

KSOS Final Report

few subtle or elusive bugs have been discovered.

The lowest level routines were driven by test frames and their behavior checked. Layers of routines were successively added and checked at each stage until a module, major section, or the whole Emulator could be tested. Since during most of the Emulator implementation effort the Kernel was incomplete or unstable, other methods were devised to be able to test the Emulator on the PWB/UNIX development system.

One of these testing aids was the Kernel Simulator. This tool was developed originally by the Emulator team and later extended and enhanced by the Network implementation team. By simply substituting EMT for TRAP instructions, a user program and Emulator could be loaded together in a single process, with the Kernel Simulator running in a sibling process. Using the UNIX ptrace facility to examine and change the data space of the Emulator, the Kernel Simulator could mimic the operation of the Kernel for selected Kernel calls.

The Kernel Simulator worked for single process testing. For the testing of multiprocess interactions a package of functions was written which permitted coroutines in C. Also, a set of functions for emulating certain Kernel calls was written. These Kernel Emulator functions would call upon the coroutine mechanism whenever a "Kernel" process switch was appropriate. This permitted a fair amount of confidence to be gained in critical concurrent code before testing was undertaken on the live Kernel.

Due to the stark nature of the live Kernel environment, all the debugging that could be accomplished on PWB/UNIX was effort well spent. Other than the debug code built into the Emulator, the only useful debugging aid in this environment was a trace facility built into the Kernel. The detail of the trace was selectable from the console switches.

5.2.8 Conclusion

Although the Emulator implementation was not totally completed, enough of the functionality was present to run a number of UNIX programs, including the shell. Unfortunately, the Emulator is large and slow. This is at least partially due to the fact that the processing of user requests is split between two domains. It is also due to the overhead of the multiprocess interactions within the Emulator. An alternative architecture which has some promise is to have the multiple process abstraction of the UNIX process family in the user domain be created by the Emulator residing in a single Kernel process in supervisor domain. This would require some changes to the Kernel in the form of number of processes and number of segments per process supported, as well as some tuning of the Kernel interface functionality to the needs of the Emulator.

5.3 NKSR

5.3.1 Introduction

The Non-Kernel, Security-Related software (NKSR) is a set of programs which provide KSOS with the necessary operating system support. The NKSR provides functions which are basic to the system (e.g., essential for further program development on the KSOS Kernel), functions necessary in a multi-level secure environment, and functions necessary to support a general purpose operating system.

The NKSR is divided into four classes: Secure User Services, System Operation Services, System Maintenance Services, and System Administration Services. These classes contained the following functions:

1. **Secure User Services**—This class contains functions that initialize the secure environment and provide a secure path from the user to all of the trusted NKSR services. Some examples of programs in this class are:
 - **Initial Process**—The Initial Process is the first process to run on KSOS after the boot procedures have completed. This process initializes the security level of system objects and invokes the Secure Initiator.
 - **Secure Initiator**—The Secure Initiator spawns the Audit Capture Process (see System Administration Services below), builds the Process Bootstrapper and spawns a Secure Server Process for every configured terminal.

KSOS Final Report

- **Secure Server**—The Secure Server process provides secure access to NKSR services. This is achieved by having a dedicated character that is recognized by the KSOS Kernel as the Secure Attention Character. The Secure Server has four basic states. Initially it is in login state which only allows access to a single NKSR function, namely login. When Secure Attention is input the Secure Server goes into dispatch state. In this state the name of the NKSR function is input and the Secure Server will dispatch that function. When a NKSR function is active the Secure Server is in a NKSR function wait state. Note that Secure Attention is still recognized in this state, however no NKSR functions that are performed external to the Secure Server may be dispatched until the previous NKSR function has completed. The remaining state occurs when a user process is running and the Secure Server is waiting for Secure Attention.
 - **Login**—The Login process performs user authentication functions and creates a user's default initial environment. This process is available to all users and is accessible via the Secure Server.
 - **Logout**—The Logout process destroys all environments and processes of the user. After logout the Secure Server returns to login state.
 - **File Access Modifier**—The File Access Modifier allows users to change the type independent information of a file. This permits users to change a file's security and integrity levels. Monitoring of this process is performed and security downgrading can be restricted to the system security officer.
 - **Level Preserving Copy and Print**—These utilities copied and printed files while protecting against accidental security upgrades. KSOS, by implementing a protection policy similar to the Bell and LaPadula model, allows for information to be freely read upward in security. To inhibit material from migrating to system high, Level Preserving Copy and Print were provided.
2. **System Operation Services**—This class contains functions that are necessary to support a general purpose operating system. Some of the programs contained in this class must perform functions for several different users potentially simultaneously. This requires multiplexing information from various users while preserving security levels. Many similar functions in UNIX allow serious breaches of security to occur. The essence of many violations is that the user is able to write on a file owned by someone else, for example a mailbox. In KSOS, Users are prohibited from such direct actions. Rather, intermediate files inaccessible to the sender are used. In this way, an important class of UNIX security flaws is systematically precluded.
- Some of the programs contained in this class are:
- **Line Printer Spooling**—The line printer spooler mechanism provides a secure method of spooling and printing files. The line printer daemon is the only process that may write to the printer. This allows the daemon to correctly enforce the system security policies. When KSOS is booted, the line printer daemon will remain off-line until the operator sets the maximum security level to be printed. Once in operation the daemon is responsive to a set of operator commands to: stop, start, remove a file from the queue, etc. Such control over the daemon is essential in a secure environment, killing a daemon in an unknown state, ala UNIX, is unacceptable.
 - **Secure Mail**—The mailing mechanism allows multi-level mail to be sent and received. The mechanism was not trusted, i.e., it did not violate any of the Kernel's protection policy, and executed on the UNIX emulator.
 - **Mount/Unmount**—The mount and unmount mechanisms allow users to mount file systems securely. These programs assure that the file systems have been immigrated into this KSOS system and that the user had the proper security access to perform the operation.
 - **Assign/Deassign**—These functions allow users to securely share devices, such as tape drives and paper tape readers, in a multi-level system.
3. **System Maintenance Services**—This class provides the necessary programs to set up and maintain the KSOS file system structure and perform other necessary system maintenance functions. A fundamental problem in the area of system maintenance is the existence of a user who is not bound by the security rules of the system. This "superuser" normally can access any file, execute any program, kill any process, etc. Such untrammelled power is unacceptable in a secure system. In particular, because this

KSOS Final Report

power is associated with a particular user, it is very vulnerable to Trojan Horse attacks. (A normal user can create a program that if ever executed by the superuser will create an exploitable security breach.)

In KSOS a different philosophy was followed. Instead of a privileged user running ordinary programs to do system maintenance, individual programs are given various privileges. The invocation of these programs is further restricted to authorized users, using the standard protection mechanisms of the system. The unanswered question about this philosophy is whether this suite of trusted maintenance programs is adequate for the long term operation of the system. Based on the substantial experience of several members of the KSOS team and the combined experience of operating and maintaining the (15+) UNIX systems in operation at FACC during the project, it is believed that the suite of NKSR maintenance programs provides sufficient functionality for normal operation and maintenance.

This class contains programs such as:

- **Dump/Restore**—These programs allow the dumping and restoring of known, i.e. immigrated, file systems on KSOS. This facility provides the system with basic file system backup and migration facilities.
 - **File System Checking**—This facility consists of programs which check KSOS and UNIX style file systems.
 - **File System Maintenance**—This facility consists of programs which allow for file system initialization and maintenance.
4. **System Administration Services**— This class of NKSR services includes programs which assisted the system administrator in setting up and maintaining a multi-user, multi-level system. Some of these programs are relatively straightforward, e.g., simple editors to edit the system security related data bases. However, such editors are required to avoid the potential Trojan Horse security threats that would be caused by using a non-NKSR editor.

This class contains programs such as:

- **User Installation and Removal**— These functions allow the administrator to easily install and remove system users. This involves setting the user's minimum and maximum security and integrity levels, establishing the user's default login environment (process), providing the user with a password, etc.
- **File System Immigration**— This function allows file systems to be migrated from one KSOS system to another. It also provides programs to migrate UNIX files to a KSOS system.
- **Audit Capture**— This audit mechanism provided the basic information gathering and reporting system that is required for secure system operation. The messages generated at various times by the Kernel and the NKSR are collected by the Audit Capture Process and written to an audit file. No reduction or analysis of this data was attempted by the delivered KSOS software.

5.3.2 Technical Development Experience

The NKSR contained contained technical issues and problems which were new to the operating system environment. Most of these issues resulted from the fact that KSOS was a multi-level secure system.

5.3.2.1 Noteworthy Accomplishments

The NKSR area had many technical accomplishments which were realized as system policy, NKSR programs, and Kernel changes.

The capability for exclusive use of system objects was provided by the KSOS Kernel to support NKSR programs such as the Directory Manager and the File Access Modifier. KSOS objects could be opened for read, write, read_write, exclusive write and exclusive read_write. An exclusive open would only succeed if there were no other opens in the corresponding mode (read, write). The Directory Manager, which performs all directory modification functions, needed to open directories for exclusive write. By opening the directory in such a manner, the directory manager process would lock out any other directory manager processes from simultaneously modifying the same directory, however other processes would still be allowed

KSOS Final Report

to read the directory. This scheme provided for efficient and safe directory management.

The File Access Modifier changes the type independent information of a file. In order to change the level of the file securely, the File Access Modifier must have exclusive control over the file. Namely, when changing the security level of a file, the File Access Modifier would open a file for exclusive read-write (locking out other potential readers and writers of the file), display the file, change the security level of the file, and then close the file. This procedure guarantees that the file that was viewed is the same file that had its security level changed.

The concept that objects had subtypes was also generated as a result of NKSR needs. KSOS objects had both a type (e.g., file, extent), and a subtype (e.g., directory file, swap extent). The need for subtypes was generated to support the Directory Manager. Some Kernel enforced control was needed to ensure that non-directory manager processes could not modify directories. By having directories be files with directory subtype, the Kernel could disallow directory modifications unless the process had the directory subtype open for writing. Then by restricting access to the directory subtype so that only the directory manager could open it for writing, directories were given the required protection.

Subtypes were generally used to give added protection to system objects that could not be achieved by other means, such as discretionary access or security. Subtypes were used to mark extent types, e.g., swap extent, KSOS file system extent, etc., and to designate special files, such as directories and network files.

Some of the NKSR software was contained within the boundaries of the KSOS protection model. Often, however, to provide the needed service, it was necessary for the NKSR programs to violate the KSOS protection model in defined and controlled manners. Nonetheless, violation of the protection model was always avoided whenever possible. For example, to provide the services of mailing and spooling it was necessary to change the owner of the data from the originator to that of the service manager. A method of changing the owner was used that did not violate the KSOS protection model. Namely, the originator would start the service process, which would in turn start a process which would run as the service manager. These two processes would then share an interprocess communication channel. The data would be read by the owner of the data, given away (by writing on the shared communication channel) to the service manager, and then stored in a file owned by the service manager. In this way it was not necessary for the service manager to violate the KSOS protection model in order to read the originator's data file.

One of the major goals of KSOS, was to provide a secure channel between the user and the trusted software. This channel was to be "unspoofable". That is, it was to be impossible for any nontrusted program to intervene when the user was connected to the secure channel. This secure channel was realized by having the Kernel provide multiple I/O channels per terminal, of which only one can be active at one time. Two of these channels were used to provide secure paths to the user. When the secure attention key was hit, the Kernel would make the Secure Server path active and suspend all other paths. The Secure Server would then securely, since no other processes were allowed to perform I/O on this path, take requests from the user. If a secure service was requested, the Secure Server would start the service on the other secure path. By providing this type of terminal path management, KSOS provided a secure, unspoofable, link from the user to the trusted software.

5.3.2.2 Noteworthy Issues

5.3.2.2.1 NKSR Program Complexity

One of the objectives of the KSOS program was to produce a small and simple Kernel. While this approach provides benefits in the verification area, it impacts the complexity of Kernel based applications. In particular, the KSOS Kernel exported many functionalities normally performed by an operating system Kernel, such as: directory management, mounting/unmounting file systems, process creation, etc. The NKSR programs that provided these functionalities were orders of magnitude greater in size and complexity than similar Kernel functions would have been.

5.3.3 Security Experience

KSOS attempted to utilize ideas that were on the frontier of secure systems technology. Since much of KSOS's security policy was implemented by the NKSR, much of this new technology was realized by NKSR programs.

KSOS Final Report

5.3.3.1 Noteworthy Accomplishments

Protection had to be provided for non-mountable devices, e.g., tape and paper tape. Since these devices were shared among the system users, the problem was to allow sharing of the device in a manner that was consistent with the KSOS protection model. The devices could not exist at a single level, i.e., having the tape drive at [System High Security, All Compartments] would allow all users to write to the tape, however it would not allow all users to read the tape. KSOS solved this problem as follows: Dormant devices were owned by the system resource manager. When a user wished to use a particular device, the user would request the secure service Assign to assign that device to the user. If the device was not in use, i.e., dormant, then Assign would change the type independent information, security level, integrity level, and owner, of the device to match that of the user. This would allow the user to access the device in any manner consistent with the KSOS protection model. When the user was finished with the device, he would request from the secure service Deassign that the device be deassigned. To ensure security, a tape drive was unloaded when deassigned.

KSOS provided a mechanism to safely change the level of a user readable file. This was accomplished using the Kernel feature of exclusive I/O to a system object. As stated in the above technical section, the File Access Modifier program would open the object for exclusive use, display the file, change the level of the file, and then close the file. This guarantees that the file viewed is the same file that was changed in level. Note, however, that this approach did not permit other trusted programs to scan the file to assist in the classification change.

Most operating systems run programs as the user that invoked the requested program. UNIX provided the "set user ID on execution" feature, so that users could execute programs as another user. On KSOS, because of the added security and integrity constraints, it was necessary to greatly expand the manner in which programs were set up for execution. Some programs had to execute at the security level of the requesting user, others had to execute at some fixed level, still others had to execute at a level relative to the invoking user, or to one of the program parameters. Since it was desired to restrict the privilege to change level, these programs had to begin execution at the proper security level. This was accomplished by having the Secure Server consult a data base which indicated the levels at which the programs were to be initiated. The Secure Server, using a trusted process bootstrapper, would create these processes at the proper level. This was the simplest solution. Perhaps a better solution, although more expensive in terms of system space, would be to have the Kernel associate the level of execution with each executable object.

5.3.3.2 Noteworthy Issues

5.3.3.2.1 NKSR Verification Difficulties

KSOS achieved little verification assurance of the security of the NKSR software. The security of KSOS is provided by the Kernel and the security related NKSR utilities. These NKSR utilities, in order to provide the required service, were not bound by the Kernel's protection policy, i.e., they had privileges. The security model of KSOS, from the verification perspective, only addressed the Kernel enforced protection. No model existed for the NKSR portion of the KSOS protection environment. The lack of an NKSR security model greatly impacted the verification effort. However, it is not clear that the currently accepted security model is adequate to describe the security properties of general purpose multi-level secure operating systems.

5.3.3.2.2 Integrity

The protection policy provided by KSOS included an integrity aspect. The integrity model [Biba], defined a partial ordering of integrity levels, which included a classification and compartment set. KSOS used only the classification portion of the model. The integrity classifications were used to separate different levels of system software. That is, the system software was partitioned into: user accessible, operator accessible, and administrator accessible programs. This approach provided half of the protection desired for system software, however, it did cause problems. Almost all the high integrity programs (which comprised over half of the NKSR software) were required to violate the integrity model in order to read data at lower security levels, e.g., the file system checker needed to read the file system, which violated the *-integrity rule. In at least one case the simple integrity rule had to be violated. Namely, the Secure Server executed at low integrity so that it could receive process termination IPC's from low integrity processes such as an Emulator. However, it was also required to write high integrity data bases, thereby violating the simple

KSOS Final Report

integrity rule.

It became apparent that applying the integrity model to operational protection systems needed more thought. It is now clear that the integrity compartments should not have been deleted from the KSOS protection system. They could have been used to protect system objects that spanned security levels, e.g., directories.

5.3.3.2.3 Powerful Tools

KSOS did succeed in providing a set of utilities that eliminated the existence of a superuser. However some of the KSOS utilities might be considered too powerful for a secure environment. For example, the System Administrator is responsible for administrative and security related matters, such as: adding and deleting users, changing levels of files, etc. The System Operator was responsible for the smooth operation of the system. This included tasks such as file system immigration, file system correction, etc. The problem is that the Operator had access to tools which gave him the ability to perform some of the Administrator functions, such as changing the security level of a file. Specifically, the Operator's file system maintenance tool, MCE, could change the security level of a file, reassign the blocks from one file to another, and other such functions. Either less powerful tools must be provided for the Operator, or the integrity distinction between the Operator and the Administrator should be removed.

5.3.3.2.4 Privileges and Verification

The KSOS NKSR was built according to the "rule of least privilege." That is, NKSR programs were only granted the minimal required privilege set. This approach increased the complexity of the verification process because the entire program had privileges and therefore the entire program had to be verified. A better approach would have been to give all the NKSR programs the privilege to modify privileges. In this way the NKSR programs could surround the critical privileged section with enable and disable privilege set commands. This would reduce the scope of the verification effort from the entire program to a critical section of code within the program.

5.3.4 Conclusion

The purpose of the NKSR was to supplement the functionality provided by the KSOS Kernel in order to provide a general purpose multi-level secure operating system. Toward this end the NKSR was quite successful. However, the development did reveal new information concerning secure systems development, such as: the importance of a complete system security model, the need to further analyze the inclusion of integrity within protection models, and the trade-off between including or exporting functionality from the Kernel. Considering that this was the first version of the KSOS operating system, the current collection of NKSR software is a sound and complete system support software base.

5.4 Network

5.4.1 Approach

The KSOS network software employs a distributed protocol handling. The structure of the protocol layers is shown in Figure 5-1 (where: FTP = File Transfer Protocol, TCP = Transmission Control Protocol, IP = Internet Protocol, and ARPA = ARPANET Protocol).

KSOS Final Report

```

*-----*
|  FTP  |  etc.
*-----*

*-----*
|  TCP  |
*-----*

*-----*
|  IP   |
*-----*

*-----*
| ARPA- |
|  NET  |
*-----*

```

Figure 5-1

The original intent is shown in Figure 5-2.

```

*-----*
|  FTP  |
|  etc. |
*-----*
/ > USER

*-----*
|  TCP  |
*-----*
/ > EMULATOR

*-----*
|  IP   |
*-----*
/ > SUPERVISOR

*-----*
|  ARPA |
*-----*
/ > (NKSR) DAEMON

*-----*
| DEVICE |
| DRIVER |
*-----*
/ > KERNEL

```

Figure 5-2

The distribution of functionality exists primarily for reasons of verifiability. The TCP and IP protocol handlers were deemed unverifiable, and thus could not be part of NKSR. One of the main design goals was that the software be upgradable to multilevel secure operation, but initially only a single security level network was envisioned. Unfortunately, IP does not contain sufficient routing information to identify the process issuing the request. This constraint resulted in the structure shown in Figure 5-3.

KSOS Final Report

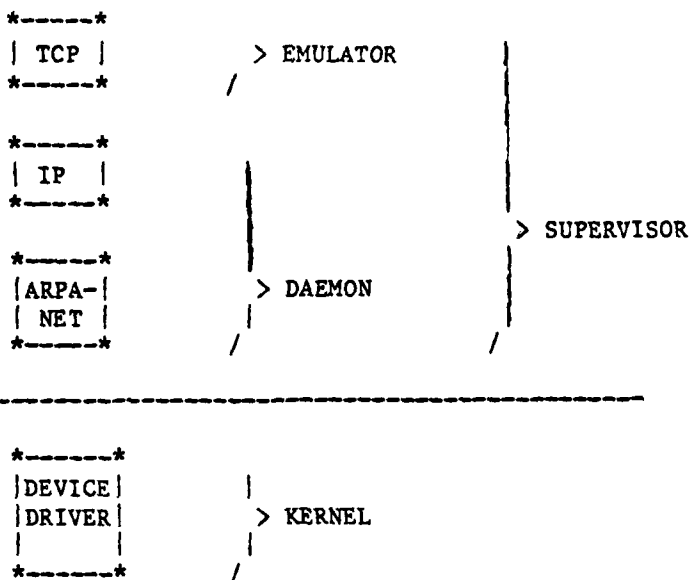


Figure 5-3

In retrospect, the security considerations may have precluded (correctly or incorrectly) a promising alternative which is portrayed in Figure 5-4.

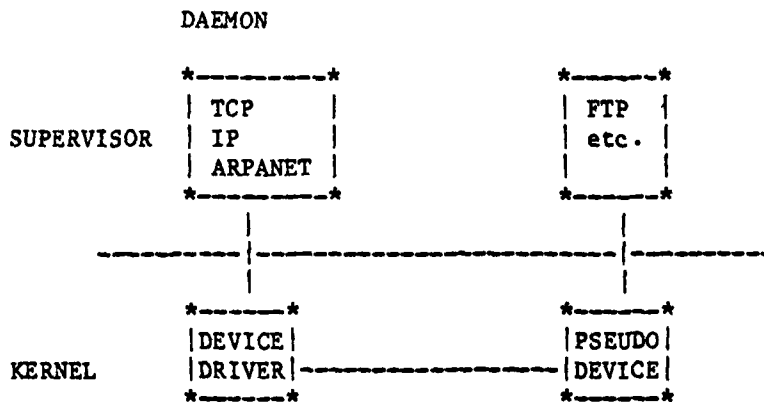


Figure 5-4

The current scheme is illustrated below in Figure 5-5.

KSOS Final Report

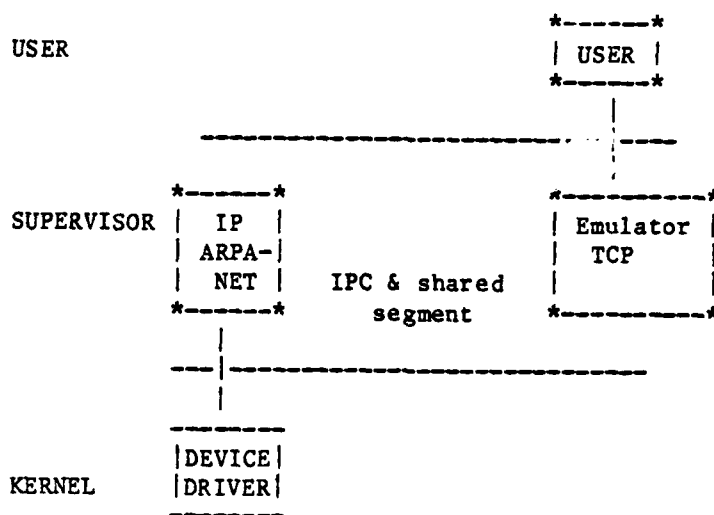


Figure 5-5

This scheme required extensive use of IPC to communicate with the Daemon. Although all data transfer was via a shared segment, there is now evidence that the scheme will be unacceptably slow. Since the KSOS Kernel does not provide any synchronization or mutual exclusion primitives for shared objects, there was additional necessary IPC traffic and additional complexity in TCP to control the access to the shared segment.

Another problem was that, in the final scheme, the user interface to TCP was via UNIX calls to the Emulator. TCP requires asynchronous I/O requests, and the ability to receive notification of asynchronous events (URGENT signal). Unfortunately, this does not map at all well onto the UNIX I/O calls. Extensions were proposed to allow asynchronous I/O calls but were hardly a desirable solution.

5.4.2 Protocol

While the protocol definitions did not finalize early in the project, the only changes were trivial ones (addition of options, etc.) which did not substantially affect the design. The problem with routing seems to have derived from a (perhaps groundless) anticipation that the protocol people would provide facilities in TCP & IP which were friendly to the requirements of the security community. Unfortunately, the architecture of secure networks is not yet an issue upon which there is a consensus. Furthermore, the addition of a security option does not constitute a solution to the problems of multilevel security for networks. Consider, for example, the problems of limiting access via networks. There was substantial mechanism in KSOS (secure attention key, etc.) to ensure unspoofable access to the system, but this was not really supported for network access (and IP/TCP still does not provide enough assistance in this area.)

The real design problems stemmed from the following considerations:

1. Since the Emulator is essentially a distributed operating system there are difficult problems associated with maintaining shared data structures. This is largely due to the design of the Kernel (or to design of the Emulator—although the decision to distribute was based on limitations on the number of segments in the Kernel and an early commitment to having one Kernel process per UNIX process.)
2. Each TCP connection (like any I/O device or file) is potentially shared between all the instances of Emulators in a process family. Therefore there were severe problems in maintaining the data structure (shared) for a connection.
3. Since communication to the Daemon was via IPC, one process in the family has to be nominated as the recipient of IPCs from the the Daemon for any particular connection (the "connection master"). This leads to problems such as: what happens when the connection master dies? (Other processes in

KSOS Final Report

the family may still have the connection open!) While this part was never really designed fully, it must be recognized that there is substantial potential for timing problems, such as in the determination of which process is the new connection master. The Emulator had similar problems with asynchronous terminal I/O, which is far simpler than TCP.

5.4.3 Implementation Problems

The lack of a reliable Kernel during implementation was a severe problem. There was no hope of doing serious implementation in the absence of a Kernel, although some effort was finally invested in the Kernel simulator, which permitted something to be done. Of course, TCP required the Emulator and the Emulator required the Kernel. It was perhaps unreasonable to expect that these three development efforts could proceed in parallel and be integrated into a fully functioning system without considerable difficulty in the final stages. Even when a fairly stable Kernel, and a slightly debugged Emulator existed (i.e., near the end of the project), trying to debug multiple process interactions on KSOS was extremely difficult. More serious advance consideration should have been given to the problems of debugging, diagnostics, etc.

Since, in retrospect, verifiability was an unsatisfiable goal within the scope of this project, the use of verifiability/non-verifiability as a design consideration was fruitless. In the light of the difficulties of debugging on top of an unfriendly system with no significant aids, the only criteria of any importance to the design should have been simplicity. The network software was anything but simple.

Finally, the Network software was complex, as was the Emulator, and TCP would probably not have fit in a full Emulator. It is not presently known how much would have had to be thrown out of the Emulator to get TCP in, but it is now clear that the Network would never have reached its initial goals.

While no timing studies were ever done on how long it took for an IPC exchange between two processes, it is probably not fast enough to support the large IPC traffic which would have been necessary for TCP. Telnet, for example, operates in single character mode, and with three or more IPCs per character, the system performance degradation would have to be considerable.

5.4.4 Postscript

Some additional comments on the design follow:

The Daemon actually had to have a fairly extensive knowledge of TCP headers to reroute packets to the appropriate process, and also to handle the closing of connections, so the hoped-for separation was more imaginary than real.

The Daemon had to perform storage allocation on shared segments for outgoing/incoming packets, yet the flow control aspects of the protocols were embedded in the TCP layer, which was handled by the Emulator. Thus, in this respect, the structure was incorrect.

6. METHODOLOGY

6.1 Languages

6.1.1 Implementation Language Selection

The implementation of any program depends in subtle ways upon the programming language chosen for the task. The computational model provided by the language affects the way in which the implementors view their system. It provides not only the vocabulary of primitives from which the system is constructed but also the vocabulary of concepts in terms of which the implementors cast their communication with the system designers, with its users, and with one another.

Nowhere is the choice of programming language so critical as it is in the case of an operating system. To be useful, an operating system must, itself, establish and maintain an efficient and consistent computational model for use in unforeseen ways over a long lifetime. In addition to these already stringent operating system requirements, the KSOS operating system is required to provide strict enforcement of a security model and must be shown to conform in detail with a body of formal specifications. These added goals of security and provability also depend heavily upon proper choice of programming language.

Both the customer and the contractor anticipated the importance of a suitable language to the success of KSOS. Criteria for language selection were developed early and published in the KSOS Verification Plan. Implementation language was identified at the PDR as an element of technical risk to the program. The identification was only too accurate; the language problem has required almost continuous attention on the part of KSOS program management and technical personnel.

A decision procedure for language selection was also developed during Phase I and published as part of the Verification Plan. This procedure has served KSOS well and continues to be the operational rule-of-thumb for language selection and is therefore reproduced here verbatim from the Verification Plan, p. 40.

IF the Euclid compiler development effort at Toronto continues on its present course and gives adequate support THEN use it—with some accommodation for hardware error signaling. Some sub-setting of the language may be desirable, for simplicity, efficiency, and enhancement of any eventual verification.

ELSE IF Modula has been adequately extended and those extensions supported, THEN use it.

{At present [March 1978] it seems that Euclid will be appropriate, although Modula provides an attractive alternative. At this point it is extremely unlikely that any more ELSE clauses are needed. However, there are other languages that may be appropriate, e.g.,}

ELSE IF ILPL is adequately supported, THEN use it.

ELSE IF Gypsy is adequately supported, THEN use it.

ELSE use UCLA Pascal.

{It is remotely feasible to use C, but only with the addition of various programming constraints, and with the explicit understanding that the kernel would subsequently be recoded in a more suitable language, before any program verification is attempted. However, such a course is neither necessary nor advantageous.}

In December 1978, LanguageGram #2 made the following observations:

- The Euclid compiler development at Toronto was unable to keep to its schedule. The language was unsupported.
- +++ Modula was well supported. Extensions were thought to be, for the most part, easy to accomplish.
- ILPL was not supported.
- Gypsy support was not yet adequate.
- +++ UCLA Pascal was available.

It seemed clear from the above that Modula should be the KSOS implementation language, and that

KSOS Final Report

UCLA Pascal should be prepared to serve as a contingency fallback language.

With the success of KSOS depending so heavily upon the proper choice of implementation language, and with the difficult experiences with Euclid fresh in mind, we undertook critical examination and comparison of Modula and of UCLA Pascal. LanguageGram #2 undertook to examine the languages, describing the criteria upon which they were examined, gave criterion-by-criterion results of the examination, and finally drew the conclusion that the selection of Modula as the KSOS implementation language was well justified at the time.

6.1.1.1 Candidate Languages

Modula and UCLA Pascal have a common ancestor: Pascal. The chief aim of the Pascal author was to develop a language for the teaching of programming as a systematic discipline. For this, they based Pascal upon the principles of structuring and the expression form of Algol 60. They replaced Algol 60's declaration with one of their own devising and incorporated data structuring facilities. The language was widely used and there existed a great number of implementations. Much of "modern" programming language development had been based upon Pascal. There was a Pascal Users Group, and a committee working toward ANSI Standard Pascal.

6.1.1.1.1 Modula

Modula is a later product of the Pascal authors and is largely based upon that earlier language. The intention of the Modula authors was to define a language well suited to the programming of dedicated computer systems, and to this end they provided facilities in the language for multiprogramming and for the operation of peripheral devices. These facilities are inherently machine-dependent. The language therefore provides a construct, called a *module*, to encapsulate them (i.e. to restrict their validity or existence to a specific and small section of the program). Another goal of the Modula authors was that the language should be small and easy to implement.

The Modula implementation used for language evaluation was obtained from the University of York (England). The compiler obtained ran under UNIX, and was slightly modified to run under PWB/UNIX.

6.1.1.1.2 UCLA Pascal

UCLA Pascal was described as a systems programming subset of Pascal. It was developed as an implementation language for the Data Secure Unix project.

The differences between UCLA Pascal and standard Pascal in terms of language features are as follows: UCLA Pascal has no WITH, no SETS, no variant records, and no real arithmetic functions (although real variables are supported). Standard Pascal I/O (files, etc.) is not supported. Although NEW is not supported, pointers are. UCLA Pascal adds declaration of variables at fixed locations, the passing arrays of various sizes to a single procedure, and extensive bit operators.

UCLA Pascal has been extended by the Systems Development Corporation (SDC) who utilize it for maintenance and extension of the UCLA Data Security Kernel. The major SDC additions are register variables, yet more bit operators, procedures and functions as formal parameters, multidimensioned arrays, and the C ++ and -- operators.

The language is implemented as a translator from UCLA Pascal to C. (The standard C compiler is then used to complete the compilation.) The version used for evaluation was obtained from SDC, and will be referred to as Extended UCLA Pascal.

6.1.1.2 Evaluation Criteria

The criteria for the selection of an implementation language for KSOS flowed directly from the goals of the KSOS project. It was required that the KSOS implementation language contribute in as many ways as possible to the security of the system, to the correctness of implementation, to the ease of coding, to the understandability of the programs, and to the ease of verification. The chief criteria for language selection were these:

- The language must exist. Its definition must be precise and there must be a compiler for it which runs on and produces code for the KSOS target machine (PDP 11/70).

KSOS Final Report

- The compiler must be supported in such a way that repairs, extensions, and modifications required for KSOS implementation and maintenance can be made quickly and effectively.
- The compiler should produce code which can run on a bare machine. (After all, KSOS will have to.)
- The object code produced by the compiler should be efficient in its use of machine resources. (I.e., small and fast!)
- The language should allow user-defined types.
- The language should be strongly typed and type-safe.
- The language should provide for encapsulation of objects. This is seen as an aid to correct programming and as an aid to verification.
- Multiprogramming must be supportable.
- The writing of machine dependent pieces of code must be supported. Ideally the machine dependent pieces can be encapsulated.
- The language, or a minor restriction of the language, should be verifiable. Ideally, a verification environment for the language would exist.
- The language should be supported by development, testing and debugging tools.

All of these criteria are important in selecting a language for KSOS, but we neither anticipated nor required that any single language meet all of them. Also, other valid criteria for language selection could be nominated. We felt that these comprised a fairly complete list of those which both were of primary importance to KSOS and had the power to illuminate the differences between Modula and Extended UCLA Pascal.

6.1.1.3 Comparison

This section contains the conclusions of a criterion-by-criterion comparison of York Modula and Extended UCLA Pascal. The criteria and conclusions are those developed in LanguageGram #2.

6.1.1.3.1 Language Definition and Compiler Quality

Modula's definition is neater than that of Extended UCLA Pascal. The definition of Modula is published in the public domain; the definition of Extended UCLA Pascal is not. Modula is the product of a mature and widely respected language designer, the father of Pascal. Modula is a later product than Pascal. Modula is motivated as a language, Extended UCLA Pascal is apparently not. The Modula compiler does a better job than the Extended UCLA Pascal compiler of protecting the language user from the mysteries and mechanisms of the compilation process. We therefore conclude that Modula is superior to Extended UCLA Pascal when judged by the criteria of language definition and compiler quality.

6.1.1.3.2 Compiler Support

Modula has an independently funded support group, Extended UCLA Pascal does not. The Modula support group is willing to make language extensions and to fix bugs. SDC is only willing to study bugs. The support of either language could be undertaken by FACC (and transferred to the KSOS maintenance contractor.) In the case of Modula, this would involve overcoming the visa problem and hiring the Modula implementor. In the case of Extended UCLA Pascal an existing or additional employee with appropriate background would be assigned to the task.

We conclude from the above that York Modula is better supported and more easily supportable than Extended UCLA Pascal.

6.1.1.3.3 Support of Bare Machine

Modula makes explicit provision for bare machine target environments; Extended UCLA Pascal does not. A Modula runtime nucleus exists. Its size is known, and it is small. An Extended UCLA Pascal runtime does not apparently exist.

We conclude that Modula is clearly superior to Extended UCLA Pascal in its support of programming for bare machines.

KSOS Final Report

6.1.1.3.4 Efficiency of Object Code

An experiment was run in order to determine the relative efficiency of the object code produced by Modula and Extended UCLA Pascal. A program for ordered insertion in a doubly-linked list was specified and programmed in both languages. The programs were compiled and the object code examined. Figure 6-1 gives a summary of the experiment in terms of instruction count.

Case	----- Instruction Count -----		
	Entry	Procedure Body	Exit
Modula :no pointers :WITH statements	0	73	2
Modula :no pointers :no WITH stmts	0	89	2
Extd UCLA Pascal :pointers :register vars	7	47	8
Extd UCLA Pascal :no pointers :register vars	7	68	8
Extd UCLA Pascal :no pointers :no register vars	7	95	8

Figure 6-1: Object Code Comparison.

Modula is four to eight times more efficient than Extended UCLA Pascal in procedure call entry and exit. Extended UCLA Pascal, with pointers and registers, produces much better procedure body code than Modula. Extended UCLA Pascal used without these features produces larger and less efficient code than Modula.

It must be noted that the use of pointers in KSOS programming be ruled out upon verifiability grounds. Under this restriction registers are still useful for holding iteration variables.

Given the above information, and in the absence of timings and KSOS characteristic measurements of dynamic procedure call counts, we conclude that Modula and Extended UCLA Pascal are roughly equivalent with respect to the efficiency of their object code.

6.1.1.3.5 User-Defined Types

The languages under study provide similar mechanisms for user type definition. They do however differ in type encapsulation and type safety, as will be detailed below.

6.1.1.3.6 Type Safety

The Modula type rule is explicitly stated; the Extended UCLA Pascal type rule is not. The Modula type rule is enforced; the Extended UCLA Pascal type rule is not. We conclude that York Modula is far more type safe than Extended UCLA Pascal.

KSOS Final Report

6.1.1.3.7 Encapsulation

Modula modules provide selective import and export and may be nested. Extended UCLA Pascal modules provide total break of scope and may not be nested. From the point of view of encapsulation, Modula is clearly superior to Extended UCLA Pascal.

6.1.1.3.8 Multiprogramming

Support of multiprogramming is an explicitly stated goal of Modula. It is not included in the goals of Extended UCLA Pascal (nor in those of Pascal). The Modula compiler provides facilities to support multiprogramming; the Extended UCLA Pascal compiler does not. The Modula facilities are identically those required at the lowest level of KSOS. We conclude that, in the matter of multiprogramming, Modula is the superior language.

6.1.1.3.9 Machine Dependent Code

Machine dependent facilities are built-in to Modula; they are built-on (or outside of) Extended UCLA Pascal. Both allow the use of fixed addresses within the language. Only Modula provides language-based interrupt handling facilities. Extended UCLA Pascal allows escape to external routines; Modula does not. Modula localizes machine dependent code; Extended UCLA Pascal does not. We conclude that, on balance, Modula provides a fuller and safer set of machine dependent facilities.

6.1.1.3.10 Test and Debug Tools

Some aid is preferable to none. When it comes to utilities and debugging aids, Extended UCLA Pascal is far superior to Modula.

6.1.1.3.11 Verifiability

The verifiability of Modula and of Pascal (NBS Pascal, NOT Extended UCLA Pascal) was studied on our behalf by SRI International. They were asked, for each language, to recommend programming restrictions which would be necessary to ensure the verifiability of the resulting program.

Modula has constructs which lend themselves to verification; Extended UCLA Pascal does not. The problems of enforcing verifiability restrictions on Modula are easy; enforcing these restrictions on Extended UCLA Pascal is difficult. Restricted Modula is a useful language; restricted Extended UCLA Pascal is less useful. Neither language has a verification environment, both are close to having one [or so it was believed at the time of the comparison]. The verification environment of Modula will be compatible with KSOS development methodology, that of Extended UCLA Pascal will not. We conclude that Modula lends itself to verification much more easily than Extended UCLA Pascal.

6.1.2 Hierarchical Design Language (HDL)

In the design of the various portions of KSOS, the Hierarchical Design Language (HDL) was used with the intention of its being both a design vehicle and design documentation.

The problem was that as a design evolved, it also tended to contain more and more detail. Thus, as the design converged, past versions of the HDL were superseded by new versions which more correctly embodied the overall design but which also were too detailed to be useful as design documents. This fine level of detail also made it difficult to subsequently maintain the design document.

HDL was to be used as a flowchart surrogate, and from that standpoint it was successful. It was certainly more pleasant to use than flowcharts would have been, and also permitted useful machine analysis to be performed. Our HDL processor checks syntax, maintains a symbol table, produces a cross-reference, and allows for libraries of predefined modules.

One of the drawbacks of using HDL was the necessity of translating to the implementation language. Part of the problem was, again, the level of detail to which the HDL had been written. More significant though were the differences in the constructs available in HDL and the implementation language. These differences sometimes required a nontrivial translation which provided another opportunity for errors to be introduced. This problem was even more acutely felt in the parts of the system which had not only HDL and code but also formal specifications.

KSOS Final Report

Perhaps, for the future, what would be useful is a processor which does an analysis similar to HDL but for a version of the target implementation language with a sufficiently relaxed syntax to allow the high level descriptions which HDL permits, while eliminating many of the translation drawbacks. In fact, FACC is currently investigating such an approach for ADA and ILPL as part of its R&D program.

6.2 Hierarchical Development Methodology (HDM)

KSOS was to be the first production quality secure operating system to be developed using all the development methodology necessary to support mathematical assurance of system security properties. The methodology as it was to be applied is described in a number of published documents available to the reader. This section is an assessment of how the methodology was applied, the degree to which it was successfully applied, and the accomplishments in this area that the KSOS project has achieved.

6.2.1 Assumptions

When the KSOS proposal was written, SRI International was to take an active role in the KSOS effort, helping FACC to use HDM and consulting on design discussions (with an eye toward verification). The assumption was that HDM was ready to use and that it would be made to work with any implementation language without much effort.

At the time HDM had been used for specification only, and the key gaps between HDM specification and implementation and between HDM specification and code verification were bridged only by claims that it could be done without much trouble.

6.2.2 HDM Specifications

A number of misconceptions about HDM were shared by the personnel at FACC and SRI.

1. HDM could be easily learned
2. Levels of abstraction and dependencies of external references were the same sort of thing.

The result was that when FACC personnel wrote their first formal specifications, first of UNIX and then of the Kernel interface, both sets of specifications suffered from the same problems:

1. Too many modules
2. Too many external references
3. Too many EFFECTS_OF statements
4. Not enough use of DEFINITIONS.

The reader of these early specifications was forced to chain through long series of EFFECTS_OF statements to discover the functionality of the system.

The confusion over the distinction between levels of abstraction and external references resulted in there being no attempt to provide multiple levels of abstraction in the Kernel. This seriously hurt the design. The "lower level" specifications were no more than input/output assertions for the internal procedures. Although working with consultants from SRI, the FACC team produced the early KSOS specifications suffering from this confusion. Only much later, when the chief developer of HDM came to work for FACC, and undertook to rewrite the specifications, did the seriousness of this error become clear.

SRI had a series of LISP-based tools that did parsing and type checking of SPECIAL specifications. While these were fine for small specifications, the large size of the KSOS Kernel Specs, combined with insufficient computational resources, made use of the tools essentially impossible during daylight hours. Even on an unloaded machine, 10-15 minutes of elapsed time was required to check a single large module. This problem was partially remedied by the construction (supported by IR&D funds), on the PWB/UNIX development system, of a checker for individual SPECIAL specification modules. The checker was much faster and more reliable, but could not handle the largest SPECIAL specifications. They had to be split into two or more modules. In addition, this checker did not handle all of SPECIAL, did not process mapping functions, and performed no consistency checking among different specifications. Nevertheless, it was a good product that improved FACC's ability to work with SPECIAL specifications.

KSOS Final Report

In addition to a lack of training, very little documentation was made available to FACC personnel. Only the SPECIAL manual and some papers were provided. At the time there was very little published documentation about HDM. While a handbook existed in draft form, it was not made available to FACC until several months into the design activity.

This lack of information, training, and tools made HDM seem very difficult to use, and tended to discourage the use of SPECIAL as a design medium. After producing the first draft of the Kernel specifications the project staff designed the system primarily by other means. Although the specifications were referenced, and periodic efforts were made to keep them consistent with the current design (which of course had changed considerably with time), the people trying to correct the specifications were not always those who were actually designing the system, so there was a communication problem. There were three major revisions to the formal specifications. The second revision was the most major, correcting gross errors in specification style and making them consistent with the design. The current specifications are now reasonably readable, and are believed to correspond fairly closely to the design.

In order to use HDM effectively in the design of a system the following are required:

1. Designers know SPECIAL and HDM.
2. Designers express designs and revisions in SPECIAL as they are made.
3. The other documentation and code is modified as required to conform with the specifications.
4. All design discussions center around expression of design in SPECIAL.
5. No other form of development (or documentation) is done until the specifications have become reasonably self-consistent.

These conditions would lead to a set of minor revisions, without any communications gap.

SPECIAL is intended to be used to help make design decisions. It was not successfully used in this way on the KSOS project for several reasons. First, the automated tools to accomplish this were not in place. Second, some of the verifications to be performed had never been done, even by hand. Under these conditions, it is remarkable that the effort was as successful as it was. The difficulty of the tasks was underestimated in both technical and engineering terms. Fortunately, the underestimation was not serious enough to prevent the attainment of design proofs. However, code proofs, which are at least an order of magnitude more difficult than design proofs, were a casualty. At the end of the contract the verification subcontractor still did not have a working verification-condition generator for Modula implementations of HDM designs.

The design proof approach was novel and successful. The approach was to build a formula generator based on the security model. The output would be a formula, which if true implied that any system implementing this specification is multilevel secure. The output of the formula generator could be proved by the Boyer-Moore theorem prover. Thus, a "run" at design proofs would take only a couple of hours of CPU time. The major difficulty with the approach was that it is possible to introduce as a lemma an unproved warrant about a specification. If this warrant were false, it could lead to a proof that an insecure system was secure. This actually happened once.

After a run, the output of the theorem prover was divided into proved and unproved formulas. The unproved formulas were of the most interest because they potentially indicated the security flaws. A formula could be unproved for one of several reasons:

1. A security flaw really existed in the design.
2. The formula was valid but the theorem prover wasn't powerful enough to prove it.
3. An invariant was needed to support the proof.
4. A resource exhaustion channel existed.
5. The formula involved factors not covered by the security model.

6.3 Verification

KSOS Final Report

6.3.1 Introduction

The original KSOS verification goals were the following:

1. The instantiation of the multilevel security model to SPECIAL.
2. The design and development of a computer tool (the MLS formula generator) whose input would be SPECIAL specifications, and whose output would be conjectures, the proof of which would imply that the specifications do not contain any violations of the multilevel security model.
3. The proofs that the specifications for the KSOS security perimeter (Kernel and NKSIR) conform to the security model. In the event of violations, the proof process should pinpoint the violations so that they may be eliminated or bandwidth-limited.
4. The development of support tools so that illustrative code proofs could be carried out. The goal of these code proofs is to demonstrate the feasibility of performing full code proofs at a later date.
5. The carrying out of illustrative code proofs.

KSOS has succeeded in instantiating the multilevel security model to SPECIAL and developing the MLS formula generator, in producing proofs of the Kernel specifications, in producing prototype support tools that could be used in code proofs, in producing a code proof for a version of the SMXflow module, and in producing some mapping functions (manually) showing the correspondence of VFUNs to Modula structures for certain Kernel modules. Due to a variety of organizational and technical factors indicated below, KSOS has been less than successful in producing specification proofs for the NKSIR, in producing human-engineered, fully documented support tools for code proofs, and in producing code proofs other than for simplified modules.

6.3.2 KSOS Verification Achievements

This section provides details of the achievements mentioned above, and indicates their benefits.

6.3.2.1 The MLS Formula Generator

The instantiation of the MLS model to SPECIAL, and a description of the MLS formula generator are found in the report "A Technique for Proving Specifications are Multilevel Secure", SRI CSL-109, January 1980. Although this tool has limitations, as mentioned below, the concept upon which it is based represents an important breakthrough in verifying security properties of systems. Its main virtue is that the designer of a system need only supply the specifications of the system as input; in contrast to other verification systems (e.g., INA JO), it is not necessary to supply additional assertions. Several current limitations and areas for improvement are mentioned in the above-cited report: a variety of restrictions of SPECIAL are imposed on the designer; the semantics of SPECIAL is defined only within the code for the formula generator and in the definitions and axioms of the theorem prover; the human interface is clumsy and provides little help in analyzing the output; and it is largely ad-hoc in construction and behavior creating the possibility of failed proofs that should succeed and unsound proofs in the face of security flaws. In addition to this list, we have noted in our utilization of the tool that a large majority of formulas that are sent to the theorem prover are not much more complex than $x \leq x$, which might be filtered out by a more powerful simplifier.

6.3.2.2 Proof of the Kernel Specifications

Between November 1979 and February 1980 there was intensive activity subjecting the Kernel specification to analysis using the MLS and Theorem Proving tools. The specifications for all 34 top level Kernel calls, and their supporting specifications, were processed. Space limitations prevented the entire Kernel from being processed in a single run, so the Kernel specifications were broken into 5 modules. It is significant that the modularity of the Kernel specifications permitted such a decomposition after the fact.

In the first run, 24 November 1979, a total of 1654 formulas were generated by MLS. 755 of these were sufficiently trivial so that the MLS tool was able to deduce their validity without passing them on to the Theorem Prover. Of the remaining 899 formulas, 586 were proved by the Theorem Prover, and 313 were unproved. In the final run, 5 February 1980, the figures were: 1598 formulas generated, 867 proved trivially by MLS, 416 proved by Theorem Prover, and 315 unproved. Detailed charts showing the statistics for each Kernel call are presented below. These specifications may be found in Appendix A of the Kernel Verification Results document.

KSOS Final Report

In the remainder of this section, we will discuss the significance of these numbers, and the effect that the runs had on subsequent modifications to the specifications. There is no significant correlation between the number of unproved conjectures and the degree to which the specifications contain security violations. This is because a subtle and deep violation of the security model may generate a small number of conjectures, whereas a simple and easily repairable violation may generate many unproved conjectures. Nonetheless, the totality of unproved conjectures is significant in that it maps onto the totality of violations of the model.

The first several runs pinpointed problems in the MLS tool itself and in the style of writing specifications. The MLS tool was unable to handle resource errors, renaming, and produced numerous duplicate formulas to be sent to the Theorem Prover. In terms of specification writing style, the tool had trouble with EFFECTS_OF clauses, with ordering of exceptions, and (somewhat to our amazement) with treating logically equivalent Boolean expressions equivalently (e.g. AND and OR are treated nonsymmetrically, with the result that DeMorgan's laws do not apply as far as the MLS tool is concerned). Therefore phase 1 of our efforts dealt with rewriting the specifications to work around these problems, and with the correction of certain difficulties with the MLS tool.

The next phase of our involvement with the specifications dealt with adding knowledge that the Theorem Prover would need to prove some of the unproved conjectures. For example, various security properties hold for open files, because such properties were checked at the time the file was opened, and no security changes were made since then. Such information was added in the form of axioms (or unproved lemmas), which allowed some of the unproved conjectures to be proved. However, extreme caution is needed in adding axioms (i) to avoid adding inconsistencies (in which case every conjecture is provable, due to the interesting property of logical implication, that an inconsistency implies FALSE, and FALSE implies anything), and (ii) to avoid adding consistent but unwarranted axioms. It is very unlikely that an inconsistency would be added that would not be detected subsequently by human analysis. For several runs, however, there were some unwarranted axioms to the effect that an object can only access objects at the same security level (such a system is known as "stratified"). As a result, several conjectures were proven which should not have been.

Thus, the discussion has focused on experiences with the tools themselves, and with adding supplemental knowledge in the form of axioms. The analysis of the unproved conjectures is also significant. We categorized the reasons for failing to prove a conjecture into the following:

- resource error (representing a potential channel, based on resource utilization patterns, which may be bandwidth-limited by the introduction of random delays in the implementation, rather than eliminated);
- errors that can be fixed by redesigning the Kernel;
- errors that are allowed to remain because there is no way for illicit information to leak beyond the security perimeter (including deliberate violations in the trusted software, needed to achieve desired functionality, as well as violations to hidden VFUNs);
- and valid formulas which the theorem prover could not prove.

We have analyzed every error detected by the tools, and taken appropriate action. However, funding did not permit utilization of the tools on the final version of the specifications. Thus, potentially, there are violations in the final version of the specifications that should be fixed.

Figure 6-2 and Figure 6-3 present detailed charts showing the statistics for each of the 34 Kernel calls in terms of number of formulas generated (FOR), trivially proved (TRV), proved by Theorem Prover (THM), and unproved (UNP). Following these charts, the next several subsections deal with a more detailed analysis of the changes made to the specifications as a result of analyzing the output of the tools.

KSOS Final Report

	FOR	TRV	THM	UNP
(KER1)				
K_FORK	158	39	18	101
K_GET_PROCESS_STATUS	2	0	2	0
K_INTERRUPT_RETURN	9	9	0	0
K_INVOKE	61	42	19	0
K_NAP	0	0	0	0
K_POST	10	6	4	0
K_RECEIVE	7	7	0	0
K_RELEASE_PROCESS	49	18	9	22
K_SET_PROCESS_STATUS	6	4	2	0
K_SIGNAL	6	2	4	0
K_SPAWN	149	50	31	68
K_WALK_PROCESS_TABLE	2	2	0	0
(KER2)				
K_CLOSE	29	11	15	3
K_CREATE	29	10	1	18
K_GET_FILE_STATUS	28	0	6	22
K_LINK	30	3	3	24
K_MOUNT	66	28	28	10
K_OPEN	90	18	1	71
K_SECURE_TERMINAL_LOCK	7	4	0	3
K_SET_FILE_STATUS	81	15	15	51
K_UNLINK	59	3	4	52
K_UNMOUNT	79	22	19	38
(KER3)				
K_DEVICE_FUNCTION	100	49	51	0
K_SPECIAL_FUNCTION	3	2	1	0
K_WRITE_BLOCK	211	97	114	0
(KER4)				
K_BUILD_SEGMENT	39	22	3	14
K_GET_OBJECT_LEVEL	2	0	2	0
K_GET_SEGMENT_STATUS	3	0	3	0
K_RELEASE_SEGMENT	30	15	15	0
K_REMAP	82	55	27	0
K_RENDEZVOUS_SEGMENT	49	28	3	18
K_SET_OBJECT_LEVEL	11	5	0	6
K_SET_SEGMENT_STATUS	31	16	15	0
(KER5)				
K_READ_BLOCK	309	161	148	0
TOTAL	1827	743	563	521

Figure 6-2: 11/19/79 Status of KSOS Specifications

KSOS Final Report

(The total of the last three columns may be less than the first column, due to the formula generator eliminating duplicate formulas.)

	FOR	TRV	THM	UNP
(KER1)				
K_FORK	177	96	0	34
K_GET_PROCESS_STATUS	2	0	2	0
K_INTERRUPT_RETURN	9	9	0	0
K_INVOKE	61	42	11	1
K_NAP	0	0	0	0
K_POST	8	6	1	1
K_RECEIVE	7	7	0	0
K_RELEASE_PROCESS	74	19	5	13
K_SET_PROCESS_STATUS	6	4	2	0
K_SIGNAL	6	2	3	0
K_SPAWN	149	88	11	16
K_WALK_PROCESS_TABLE	2	2	0	0
(KER2)				
K_CLOSE	48	23	0	3
K_CREATE	26	12	1	6
K_GET_FILE_STATUS	7	0	3	0
K_LINK	15	11	2	1
K_MOUNT	64	23	6	17
K_OPEN	48	27	8	4
K_SECURE_TERMINAL_LOCK	7	4	0	2
K_SET_FILE_STATUS	36	16	10	1
K_UNLINK	26	19	3	1
K_UNMOUNT	79	35	6	17
(KER3)				
K_DEVICE_FUNCTION	45	13	32	0
K_SPECIAL_FUNCTION	3	2	1	0
K_WRITE_BLOCK	211	111	100	0
(KER4)				
K_BUILD_SEGMENT	43	33	3	3
K_GET_OBJECT_LEVEL	2	0	2	0
K_GET_SEGMENT_STATUS	3	0	2	0
K_RELEASE_SEGMENT	38	17	0	6
K_REMAP	50	34	9	7
K_RENDEZVOUS_SEGMENT	48	28	12	1
K_SET_OBJECT_LEVEL	11	7	0	4
K_SET_SEGMENT_STATUS	31	16	8	0
(KER5)				
K_READ_BLOCK	254	160	94	0
TOTAL	1596	867	337	138

Figure 6-3: 2/12/80 Status of KSOS Specifications

KSOS Final Report

6.3.2.2.1 Analysis of the Specification Proofs

The value of using an automatic tool for checking conformance with a formal model of security, rather than relying on careful scrutiny by teams of humans, became obvious when the tool detected numerous "errors" that had gone undetected throughout several iterations of human inspection by the Contractor, Subcontractor, and Customer. Analysis of these errors lead to the following categorizations:

1. errors that could be removed by providing additional information or by syntactically reformulating the specifications;
2. errors that represent formal, but not "real", violations of the model;
3. errors that represent implicit channels, that cannot be removed without destroying needed KSOS functionality, but which can be bandwidth-limited;
4. and errors that represented wide-open security violations (there were no violations in this category).

Examples of errors that could be removed by adding or reformulating EXCEPTIONS are found in PROpost, PROreleaseProcess, PROsetprocessStatus, and FCAopen. Security violations existed in the original versions of the PRO specifications because a process could determine the (non)existence of another process at a higher level by means of an EXCEPTION value. In FCAopen, an error existed in the original specifications when a process tried to open a file at a higher level for writing (since file status information also had to be read by the process). The solution was to disallow a process from opening a file at a higher level.

An example of a violation that could be removed by Kernel redesign was in the subtype mechanism. In FCA, a global assertion was needed stating that if a process p can read or write a file f, then p can read the subtype associated with f. Thus, the level of a file is greater than or equal to the level of its associated subtype. In the original design of subtypes, there was also the rule that to write on file f, a process p must be able to write on the subtype associated with f. All the previous facts, however, imply that for any subtype x, all files associated with x must be at the same level as x. This is unacceptable in the case of directories, since users at different levels create directories. The solution was to change the above rule so that to write on file f, process p must be able to execute (not write) the subtype associated with f.

An example of a formal, but not "real", security violation is a read reference to the openCount field of FCAinfo, as occurs in FCAclose. Ostensibly this read reference is a security violation; however, the only knowledge gained is whether or not its value is 1, and if so, the file is deleted immediately.

6.3.2.2.2 Additional Lessons Learned from the Kernel Specification Proofs

In addition to the direct analysis of the output of the specification proofs, certain principles emerged which have provided useful guidelines in writing specifications, and which, in the future, may be worthwhile to incorporate in tools. Two such principles we have named the "setup principle" and the "transition principle".

6.3.2.2.2.1 The Setup Principle

The Setup Principle states: "If process p can access object o at time t, and no security-related changes occur to p or o between t and current time t', then p can access o at t' without rechecking access rights". Applications of this principle are: putting a file into the open descriptor table; and putting a segment into an address space. Utilizing the principle in a proof would involve the restriction of tranquility violations to trusted software, and proving that trusted software did not make undesirable security-related changes.

6.3.2.2.2.2 The Transition Principle

The Transition Principle states: "For finite, reusable objects (e.g., SEIDs, open descriptors), tranquility violations are acceptable if they conform to the following transition rules:

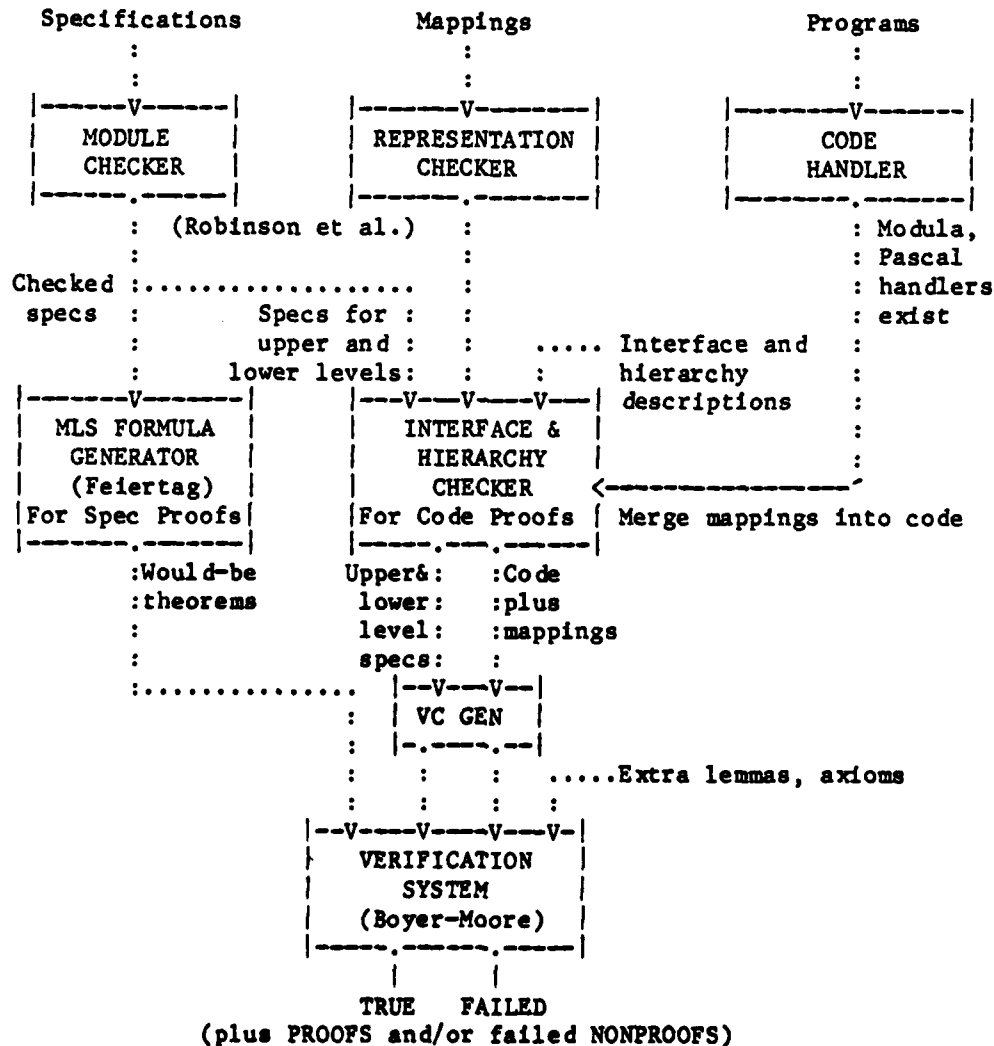
1. in making transitions from a defined to an undefined state for an object o, all VFUNs whose security level is a function of o also become undefined at the same time (VFUNs such as SENSEIDNSP (see KSOS Kernel formal specifications in CDRL-0002AF), whose security level is always system low, are thus excluded from this rule);

KSOS Final Report

2. A new object comes into existence only from the undefined state."

6.3.2.3 Prototype Tools for Specification and Code Proofs

This section includes three figures (Figure 6-4, Figure 6-5, and Figure 6-6) showing the organization of the HDM tools, the organization of the MLS proof tools, and the organization of the code proof process (including a summary and some additional notes). These diagrams reflect the status of the subcontractor's efforts at the conclusion of their participation in KSOS methodological tool development, circa the first quarter of 1980.



KSOS Final Report

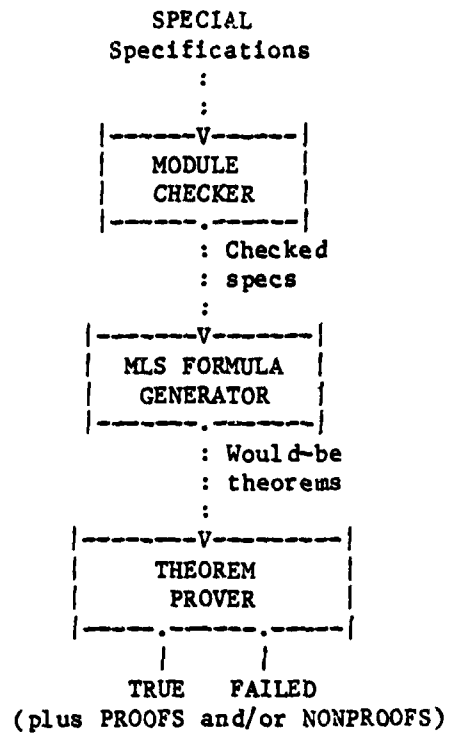


Figure 6-5
ORGANIZATION OF THE MLS SPEC PROOF TOOLS

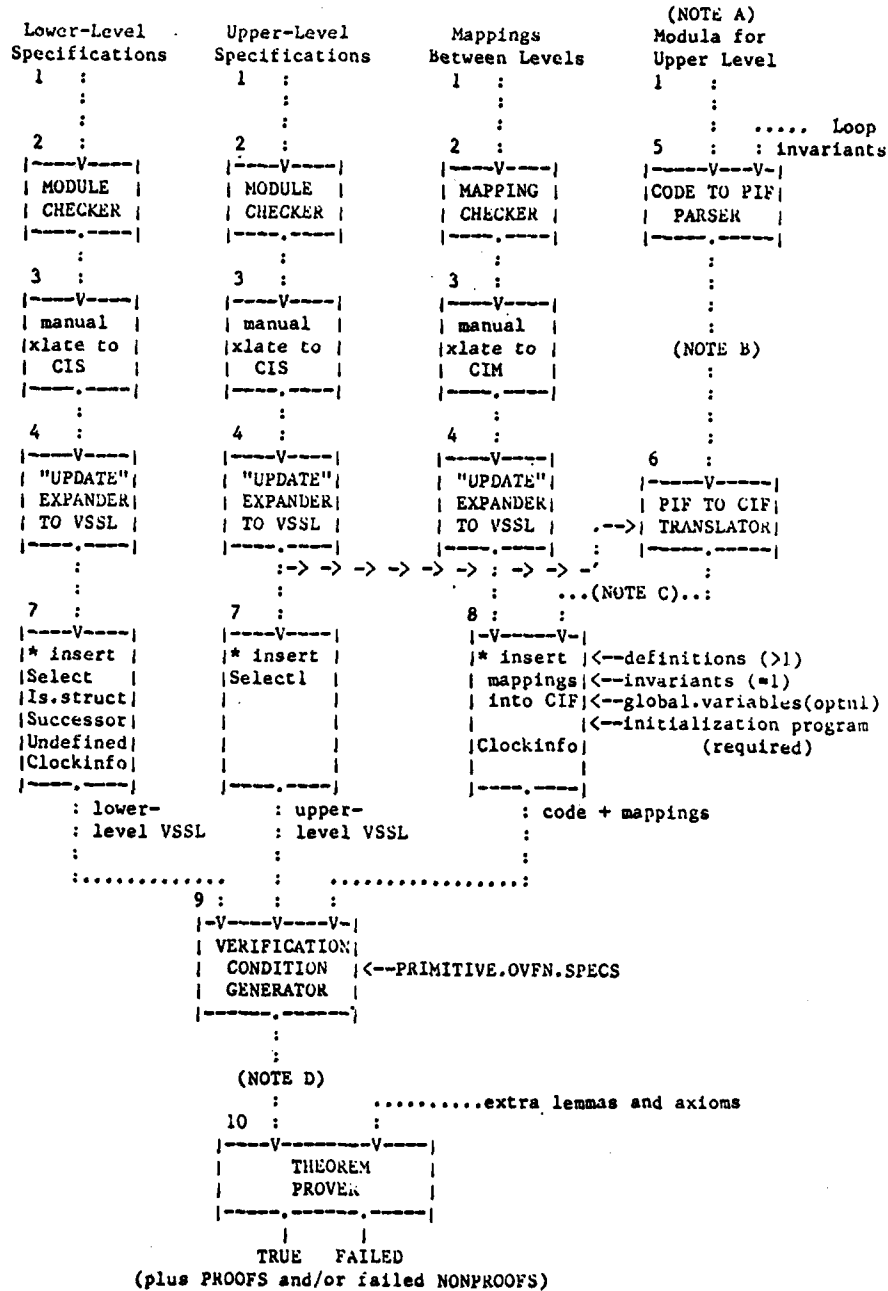
KSOS Final Report

Figure 6-6
ORGANIZATION OF THE CODE PROOF PROCESS FOR KSOS

KEY: Lower case implies human effort, UPPER CASE IMPLIES MECHANICAL EFFORT. Boxes marked with an asterisk (*) could be trivially mechanized.

6.3.2.3.1 Summary of Tools and Manual Steps

1. It is necessary before beginning the proof process that the specifications, mappings, and code conform to each other. Ideally this step is not an enormous undertaking if things are done consistently throughout. Problems that must be dealt with include various difficulties with exceptions, conflicting effects in multiple EFFECTS_OF, naming differences, different return argument conventions, etc. Special effort must also be devoted to handling sets and structures, and certain auxiliary VFUNs must also be introduced.

It should also be noted that this process is an iterative process. One of the most important aspects of this approach is that it detects inconsistencies. Thus each problem that is detected requires recycling through the appropriate paths in the figure.

2. MODULE CHECKER and MAPPING CHECKER are the HDM tools that check syntactic consistency. They have been working for four years, and are well documented.
3. Manual translation to CIS (Common Internal Specification) and CIM (Common Internal Mapping) removes quantification, accommodates structures (rewriting VFUN references in terms of SELECT, UPDATE, and MAKESTRUCT), expands nested macros, etc.
4. "UPDATE" EXPANDER translates CIS and CIM to the internal specification form (VSSL) used by the verification system. It removes all uses of UPDATE and expands them to expressions in terms of SELECT only— guaranteeing consistent manipulations of structures. Except for the structure representations, CIS/CIM and VSSL are identical.
5. CODE TO PIF PARSER parses the Modula code into a parsed internal form. This form is normally invisible to the prover.
6. PIF TO CIF TRANSLATOR translates the parsed internal form into the common internal code form used by the verification system, using the supplied (upper-level) specifications to compute exception handling instructions. CIF is documented in the Boyer-Moore HDM document.
7. Insertion of various specifications (for the upper-level SELECT, [called SELECT1], the lower-level SELECT, IS.STRUCTURE, CLOCKINFO, and the notion of UNDEFINED) is required to complete the specifications.
8. The CIF must be augmented with the mappings, CLOCKINFO, at least two definitions, an invariant (which may contain many components), global variables (which are optional), and an initialization program.
9. The VERIFICATION CONDITION GENERATOR takes upper- and lower-level specifications and code (augmented with the mappings, etc., as noted above), and generates verification conditions for the Theorem Prover.
10. The THEOREM PROVER takes verification conditions as would-be theorems and attempts to prove them. It returns either TRUE (along with the proof) or FAILED (along with its attempted proof) for each verification condition.

6.3.2.3.2 Additional Notes

- A. Before attempting any code proofs, code should have been compiled, run, and tested. These steps are omitted from this diagram.
- B. This path may be traversed either manually or automatically (in the latter case, with the ALLPARSE function).
- C. The output from the CIF translator must be manually loaded from the translator environment into the Theorem Prover environment, which are disjoint. However, this change of environment could be automated.
- D. At present the Theorem Prover must be invoked manually for the given set of verification conditions, although this could easily be done automatically.

6.3.2.3.3 Sample MLS Tool Outputs and Their Interpretation

In this section we present two sample outputs from the MLS tool: one which fails to be proven subsequently by the theorem prover, and one which is subsequently proven by the theorem prover.

6.3.2.3.3.1 A Failed Formula

This example is taken from PVMbuild, and is generated from the first effect. The relevant definitions are the following:

DEFINITIONS

```
tiiStruct proTii IS TiiInfo(pSeid);
tiiStruct segTii
  IS STRUCT(proTii.nd, ... );
seid newSegSeid
  IS SOME seid s | SENseid Nsp(s) = SENseidNsp(exampleSegmentSeid)
  AND SEGinstanceInfo(s) = ?;
```

EXCEPTIONS

...

EFFECTS

```
'TiiInfo(newSegSeid) = segTii;
...;
```

The MLS tool generates the following conjecture from this first effect, which is then fed into the theorem prover:

Proving:

(SMXcompare pSeid.1 s.1.1.1.1)

Name the conjecture *1.

Since there is nothing to induct upon, the proof has

FAILED!

The conjecture is generated based on the structural aspect of the effect, namely, that writing is occurring into TiiInfo, as evidenced by the syntax of TiiInfo being quoted. Hence, according to the multilevel security model, which allows writing to occur only in an upward or equal direction of security level, the model requires the source of the write to be less than or equal to the level of 'TiiInfo. The predicate "x is less than or equal to y in security level" is given formally by "SMXcompare(x,y)." The source of the writing is segTii, which expands from the above definitions to STRUCT(TiiInfo(pSeid), ...). The level of segTii is computed by the MLS tool to be pSeid, based on information fed to the tool at the start of the session (or remembered by the tool from a previous session). The level of TiiInfo(newSegSeid) is likewise computed by the MLS tool to be s (from the definition of newSegSeid). Hence the model requires pSeid to be less than or equal to s in security level, and generates the conjecture so stating.

The reason for the arguments of the conjecture being named pSeid.1 and s.1.1.1.1 are due to the internal workings of the MLS tool, which adds sufficient "tails" to the variable names to make them all unique. The MLS tool works in a completely flat name space.

For the above conjecture to be true, it is necessary to add more information, namely, that the level of newSegSeid is greater than or equal to the level of pSeid. This additional information can be added in a variety of ways. One way would be to add a third conjunct to the definition of newSegSeid. The information could also be added via an exception.

KSOS Final Report

6.3.2.3.3.2 A Proven Formula

Virtually all of the formulas generated by the MLS tool and subsequently proved were simple in nature, requiring substitution of variables, or propositional logic, rather than complex inductive strategies. The following example is typical.

As a first step, a global assertion stating that

`SMXcompare(SEGuseInfo(s, sd).instance, s) = TRUE`

was added to the semantics of the system, taken from an assertion in the specifications. Universal quantification is understood with respect to the parameters `s` and `sd`. The internal form of the assertion is:

```
-ADD.AXIOM(A0013 (REWRITE)
              (IMPLIES T
                (EQUAL (SMXcompare (DOT (SEGuseInfo s1 sd)
                                         (QUOTE instance))
                               s1)
                      T))
              NIL)
```

The following proof is now presented, which uses the above axiom.

```
Module: ker4
Function: K_build_segment
EXCEPTION
(EXCEPTIONS_OF (PVMbuild pSeid ss ms size vl))
```

```
Proving:
(IMPLIES (EQUAL use.1.1.1.1 (SEGuseInfo pSeid.1 s.4.1))
          (SMXcompare (DOT use.1.1.1.1 (QUOTE instance))
                      pSeid.1))
```

This formula simplifies, rewriting with A0013, to:

`(TRUE).`

6.3.2.4 Code Proof for a Simplified Module

The Modula procedure `SMXcompare` was greatly simplified for the purpose of pushing a code proof through the tools. Essentially all data abstractions were removed from the procedure and its associated specifications, and the procedure simply compared the fields of the two objects under consideration for the appropriate inequality or subset operation. Although very much a toy example, it was instructive to see the trace of the theorem prover in proving the "correctness of the implementation of `SMXCompareModule` on `PrimitiveModule`". The elapsed time was 1.065 seconds, with .124 seconds of cpu time devoted to theorem proving. This code proof may be found in the Kernel Verification Results document, Appendix B.

6.3.2.5 Manual Code/Specification Analysis

As part of the efforts leading to the final version of the Kernel specifications, we spent approximately 2 man days in manually comparing the code and specifications for the PVM module. Our goal was to get a feeling for the difficulties in carrying out a code proof. The first step was to map the types between the code and the specifications. Although there were no conceptual difficulties at this step, a variety of minor issues had to be checked. For example, `{segDes}` is mapped into `{0..15}`. We checked that no ordering properties of the integers were used at the top level in the code (where such ordering properties would be unavailable at the top level in the specifications). It was only in the implementation of the `FORALL` construct in `SPECIAL` that the ordering properties of the integers were used. Another minor example occurs in the

KSOS Final Report

mapping of `tiiStruct`. Although the mapping is virtually the identity mapping, in the specifications, owner and group are of type `INTEGER` and in the code they are of type `CARDINAL`.

The next step was to map the primitive VFUNs into the Modula data representations. There are five primitive VFUNs to be dealt with. Four of them had reasonably clean mappings. `SEginUseIndexSet`, on the other hand, had no counterpart in the code. We conjectured that its use in the specifications was to guarantee unique SEID generation in `PVMbuild` and `PVMcopySeg`. In the code, `STMGassignEntry` has the property that any SEID generated is unequal to the SEID of any existing segment. It was unclear how a mechanical code proof would account for this type of correspondence.

The next step was to map the definitions between the specifications and the code. We noted details in the code that had no counterpart in the specifications, e.g.: all virtual addresses that are base addresses for upward-growing segments are multiples of 64; no segment size exceeds $(2^{16})-512$; and all segment sizes are multiples of 512. Nevertheless, there were no major conceptual problems in this mapping.

The final step was to map the OFUNs into the Modula procedures. Our approach was to attempt "transliterating" the OFUNs into a Modula-like pseudo-code by first applying the previous mappings (and thus moving into a "Modula-like data space"), and then applying programmer's license to convert the nonprocedural aspects of `SPECIAL` into the standard sequential type of Modula procedure. Having done this, the goal was then to compare the resulting pseudo-code with actual Modula code to see if we could demonstrate equivalence. This paradigm went smoothly for the first OFUN we attempted (`PVMcreate`). For the next OFUN, however, (`PVMstore`) we encountered conceptual obstacles. In the specifications, one of the parameters was `VECTOR_OF INTEGER vec`; in the code, however, there was nothing tangible corresponding to this because `vec` represents data in transit along i/o channels or the data bus. To achieve a code proof would have necessitated resolving this problem, e.g., by formalizing appropriate aspects of the underlying computer architecture and incorporating them into the specifications. This was obviously not possible at this late stage.

6.3.3 Difficulties

As mentioned earlier, there were various goals that were not achieved. In this section we indicate difficulties which made success elusive, and where appropriate, indicate things we would do differently which might lead to greater success in the future.

1. Logistical difficulties in running verification. Getting access to the verification machine at SRI involved physically being at SRI (due to a poor communications link between FACC and SRI). Furthermore the jobs had to be run in batch mode in the evening hours. In the future, things could be improved dramatically with an on-site, accessible, interactive verification capability.
2. Tool development in isolation from applications development. A large part of the SRI subcontract for tool development and related support proceeded in virtual isolation from the KSOS development effort. In the future, a much closer relationship is called for between Contractor and Subcontractor. The design and development of tools should be coordinated with the applications that will eventually use the tools.
3. Failure to apply HDM throughout all stages of KSOS. Although the modularization achieved by the formal specifications is clean and comprehensible, there was a failure to apply the hierarchical aspects of the methodology. The original lower-level specifications were simply transliterations from the Modula code to `SPECIAL`, rather than an evolution from top-level specifications. The gap between upper- and lower-level specifications was a significant reason for failure to achieve code proofs. In the future, more rigorous use of all aspects of HDM will be required. In particular, implementation should not commence until there is a clear hierarchy of specifications, with appropriate mapping functions between adjacent levels, in which there is a reasonable gap between the bottom level and the state space corresponding to the implementation language. (It is worth noting that FACC has applied HDM in its entirety, as just mentioned, in IR&D projects during 1980 with successful results).
4. Shortcomings of Specification Technology. The lack of powerful constructs in `SPECIAL` (and all other specification languages) for concurrency and dynamic processes created another gap between the code and the specifications. For future applications, more powerful versions of HDM, in which `SPECIAL` is buttressed by appropriate concurrency and process constructs, and in which there is a closer correlation

KSOS Final Report

between HDM and the underlying theorem prover, will help achieve verification goals.

5. Oversimplicities in the basic security model. The Bell and LaPadula model which formed the basis for the KSOS security model is inappropriate in several respects, e.g., there is no direct way to model KSOS privileges (which is a major reason the NKSR specifications were not proven), and it does not allow the reuse of resources such as SEIDs (which is required in a finite implementation). In the future, more sophisticated and relevant models should be developed.
6. Technology for identifying correspondences between formal specifications and actual code. As mentioned in the previous section, methods are needed, preferably with mechanical support, for determining the correspondence. This may require implementation and specification languages carefully designed to be used together in this way, as well as disciplines for developing the two representations which keep them from diverging.

6.3.4 Conclusions

Although the use of HDM for the KSOS specification and verification effort did not accomplish all that it set out to do, it had the following significance:

1. It specified the interface of a large production system. All previous efforts were for paper systems.
2. Both the specification and verification efforts contributed to the production of the system, uncovering flaws and indicating how things could be done better.
3. It provided insights on how to build a better formal methodology for the development, specification, and verification of trusted systems.

6.4 Testing

The testing effort on KSOS was jointly carried out by the programmers and an independent test team. Both unit and integration testing were conducted by the individual programmers, while the Emulator, Kernel, and NKSR qualification and system testing were to be conducted by the test team.

6.4.1 Unit Testing

Unit testing was envisioned as an informal testing effort performed by the programmers. The testing itself was expected to be done in a top down fashion with a few tools to aid the effort. While the testing effort was fairly successful, it did not completely follow the planned scheme and some things should have been done differently.

To begin with, it appears that the unit testing effort should have been more formal. When, or in the case of KSOS, where does one unit test, for example. Much testing of the NKSR and, to some extent, the Emulator was done on the UNIX system that had to be redone on the KSOS system. Better guidelines could, possibly, have decrease the amount of wasted testing effort.

The majority of testing literature states that testing should be conducted top down using stubs to fill in lower level functions. This approach turns out not to be feasible for the KSOS project. The main reason this approach fails is that when implementing an operating system, there is no underlying functionality to support this type of testing. It is difficult, in this case, to find pieces which can be stubbed out effectively.

Two tools were developed to aid in the unit testing effort. The probe tool allows Modula code to be instrumented such that one could tell what branches were executed in a given piece of code. The "test frame", on the other hand, is a skeleton of a driver which the user can flesh out as needed.

Originally, a testing requirement for the Kernel unit tests was that all branches would be executed. The probe tool was initially developed to aid the programmer in determining which branches were executed. The tool and modified versions of it turned out to also be useful in debugging and in providing other types of information such as size information and information used in making adjustments to the Modula compiler. Perhaps the most important thing learned by this effort was that if LEX and YACC (or similar programs) are available, it is well worth the time to develop the necessary grammar. From this point, it is simple to implement many useful tools.

The concept of the "test frame" was used by the programmers to aid with the actual running of unit tests. The test frame is basically a minimally intelligent driver. It reads in a single character and, on the

KSOS Final Report

basis of this character, it branches to the section of code responsible for carrying out the command associated with that character. The section of code then reads in any necessary parameters, calls the indicated module, and prints out any returned results. This tool was particularly useful in testing the Kernel. One can, for example, test the core allocation unit by alternately allocate and release core and examine how the modules work together.

6.4.2 Integration Testing

Integration testing was planned to have approximately four major builds for the Kernel. Builds were also planned for the NKSR and the Emulator. While integration did not proceed precisely as planned, it went smoothly and without any major problems.

Originally a "build chart" was made up to indicate the order in which modules and units were to be implemented and integrated. Unfortunately, full use was not made of this chart; e.g., identifying possible bottlenecks. Also, it was not kept up to date.

The major variation of what was planned and what actually happened in the Kernel integration was that while the major builds were done, it was found that by having minor builds between the four, the whole integration process went more smoothly. The more frequent builds also allowed the NKSR and Emulator to use completed sections of the Kernel sooner.

Integration for the NKSR was done on a program basis rather than larger builds. This approach was more feasible since the majority of the NKSR programs are not dependent on each other. While the concept of a build was not useful for the NKSR, more attention should have been paid to the order in which the modules should have been implemented.

The Emulator integration became a combination of the integration techniques used by the Kernel and the NKSR. Basically there was some initial build upon which the other sections were added as they were done.

It appears that integration went so smoothly because of three main things. First and most important is the care taken to keep interfaces flexible but well defined and keeping functions the same or, in cases where it changed, the discrepancy reporting mechanism helped to keep all informed. The Modula language with its strong types also helped by providing a good mechanical check of the interfaces. Finally, having the major Kernel builds subdivided helped to spread out any problems thus making them easier to identify and handle. In general, integration of KSOS was a relatively painless process, especially when compared with the other software projects.

6.4.3 Emulator Qualification Testing

It was envisioned that the Emulator would be tested by writing small special purpose programs which would exercise the Emulator call interfaces. A goal was established to run tests in such a way that all requirements could be tested without adding any instrumentation to the system. To verify that file assign works, for example, the test would consist of checking the ability to read and write the file rather than examining an internal table to see if the file was accessible.

It was found while writing the test procedures that writing individual test programs would not be as effective as was thought. They are tedious programs to write since one has to be prepared to deal with a test not completing successfully. In the above example, one must write the code to deal with the possibility of a read failure, for example. In other words, a lot of uninteresting code would have to be written though it may never actually be used. In addition, there is frequently no way to debug the test program without running it on the system under test, which introduces the problem of using the system to test the system. This problem can make it difficult to tell if a bug lies in the system or in the test program.

It was decided that a better approach would be to write an interactive Test Language Interpreter (TLI). The TLI acts like the Unix shell except that rather than executing programs by forking off separate processes, it makes the appropriate Emulator calls directly. There is one command for each Emulator call. Not forking off separate processes avoids problems caused by the process returning; e.g., the file being opened on an open call would be closed.

It was easy to debug the TLI-Emulator interface since the TLI is an interpreter. An instruction was added to the TLI to selectively turn-on and turn-off the printing of the data presented to the Emulator by

each call. Another testing aid was the ability to run the TLI under UNIX. Since the KSOS Emulator is modeled after the UNIX interface, most of the calls could be tested in this way.

Most of the test procedures have been tested using UNIX. Some initial testing of the KSOS Emulator has been accomplished, but the PDP-11 configuration used for most KSOS testing had too little real memory to run the entire TLI system together with the KSOS under test.

The goal of being able to test all requirements without needing to instrument the Emulator was successful. All calls which changed the state of the KSOS system were able to be tested by checking the effects upon subsequent calls.

6.4.4 NKSRC CPCI Qualification Testing

The NKSRC CPCI is a collection of programs with such diverse activities as login, logoff, the system administrator functions, etc. Most of these functions are directly invoked by the operator or user, and do not require a test facility such as a test frame or test language interpreter. It was planned to test these features using only the human interface provided. There are a few NKSRC functions that don't have a direct human interface, the most notable being the network daemon. It was planned that these functions would be tested using the Test Language Interpreter developed to test the Emulator CPCI.

It appeared that the test approach was sound. The attempt to use only the man-machine interfaces when provided for testing appeared to cause no difficulties. One helpful feature for testing was that the editors which were to be used by the operator and system administrator to maintain files and data bases also could be used to deliberately introduce errors in these data bases. This ability to create errors facilitated testing the fault detection software. It also appeared that the Test Language Interpreter would be entirely satisfactory for testing the network daemon since the network daemon used the same system calls on the Emulator, but just processed them differently.

6.4.5 Kernel Qualification Testing

It was felt that the Kernel was the heart of KSOS, so a lot of effort was spent on determining how to test it. It was planned that advantage would be taken of the fact that the Kernel specifications were written in SPECIAL. It was also planned that the Kernel would be instrumented to dump its state information which would allow the test team to check that the state of the system had not changed as a result of a Kernel call which returned an exception.

In order to identify status changes that occurred as the result of a call, it appeared necessary to reformat the SPECIAL specifications. The idea was to expand the specifications of the Kernel call so as to eliminate all references to derived VFUNs, OFUNs, definitions, and the like. It appeared that it would be possible to use a general purpose macro system such as M4 processor to accomplish this. This plan, however, did not work. The basic problem is that variables have scope; i.e., a particular definition only applies to a limited fraction of the text. Therefore an expansion which doesn't take this property into account is doomed to failure. To do the expansion properly requires more machinery than the budget would allow, so other approaches had to be investigated.

Another idea which was tried and looked promising for a long time was to test the effective order in which the exceptions were tested in the code. The security model includes the order in which exceptions are tested. Changing this order may invalidate the proof. Checking the range of a variable after this variable is used to access information from a table, for example, may allow access to information which is outside the legal limits of that table. The basic idea was to create conditions which violated ordered pairs of exceptions. For example, if the exceptions were to be tested in the order ABCD, the first test condition would be chosen so that both A and B are violated. If A is tested before B, then exception A will be returned. Similarly for BC, CD, etc. Successful execution of the tests shows that A is tested before B, B before C, C before D, etc. We can then infer that the exceptions are tested in the order ABCD because of the transitive property exhibited by the order nature of exception testing. This idea becomes more complicated when it is observed that the exception conditions can't be satisfied simultaneously. The idea was dropped due to lack of time to complete the test generation process. In addition, the security model may not be an adequate model of the code from this viewpoint since the exception testing is actually embedded inside the algorithm rather than at the front of it as modeled in SPECIAL.

KSOS Final Report

After these ideas failed to pan out, we fell back to the position of generating test cases in the more traditional way directly from the textual part of the B5 specifications. This process seemed to be generating much better tests than was expected and would have been entirely satisfactory.

6.4.6 System Testing

System testing was planned to include five areas of testing: Generation and Maintenance of the KSOS system, KSOS System Operator, Performance Testing, Stress Testing, and Reliability Testing. It appeared that three tests would be sufficient to test the KSOS system. Performance testing and stress testing was collapsed into a single test, and reliability would be measured during the conduct of the system testing process and would not require a separate test.

6.4.7 Conclusion

The testing effort was not completed. Unit and integration testing were done, to some extent, on most of the completed code. The actual qualification and system testing were not started due to lack of time and budget.

While the testing effort was not completed, it was beneficial both from the standpoint of what was actually tested and what was learned. It is hoped that the things learned will be applied to future software products.

KSOS Final Report

7. RESULTS

This description of KSOS outlines the defined goals and indicates the extent to which each was met. The goals are categorized into general types. Associated with each item is a percentage representing a rough measure of the completeness of the system or the degree of attainment with respect to a certain goal. (The reader is cautioned that these numbers were arrived at by subjective estimation and are qualitative rather than quantitative; they are nonetheless believed to be useful qualitative metrics.)

Interfaces and Functionality

- Unix object code compatibility (100%). Complete compatibility exists insofar as the implementation is complete and where security constraints do not interfere.
- Direct use of the Kernel by users, including an IPC mechanism (100%).
- NKSR user interfaces, allowing user level access to security functions (100%).
- Network interfaces, both between protocols and between users and the network (40%). Most of the planned interfaces are designed, but none are operational.

Quality

- Reliability (80%). Overall, KSOS is very reliable for a new operating system. Some unfixed bugs persist, with occasional system crashes. The file system is designed to remain consistent across a system crash.
- Performance (40%). Overall performance compared to Unix appears to be a factor of 10 slower, instead of a factor of 2 as expected. However, the Kernel (as an operating system) performs functions equivalent to Unix in a factor of about 1.5. Also, NKSR functions perform in a way satisfactory to users.
- Quality assurance—reviews and testing (60%). FACC was open and responsive to project reviews and audits. Informal testing was fairly complete and documented (both unit testing and integration testing). No qualification testing occurred.

Security

- Implementation of the security model (100%).
- Formal specification (40%). Kernel interface specification was complete. Some NKSR was not formally specified. The most up-to-date (consistent with delivered system) Kernel specification was never verified, though a previous version was verified, with several security flaws being found and corrected. No NKSR specifications were verified. HDM was not fully used, in that mapping functions and abstract programs (as well as specification of lower level abstract machines) were not written. A small sample code proof of a greatly simplified part of the Kernel was done, but no real code proofs occurred.

Quantitative Limits

- Number of users and processes (100%). The system as delivered is configured for the required number of users (20) and processes (50).
- Hardware configurations (80%). All required configurations are realizable. KSOS needs substantially more physical memory than originally expected.
- Physical characteristics (memory breakdown) (40%). Both the Kernel and Emulator require essentially the full complement of the instruction and data address space available in a single domain of a PDP-11. Thus no space is available for implementing remaining functionality (such as network functions) or for extension.

Design and Construction

- Precision and thoroughness in design and implementation (60%). A great effort was made to design from formal specifications and to use hierarchical design and commenting practices. At project conclusion, each module of KSOS is described by a number of representations, none of which (in

KSOS Final Report

general) are in agreement. There is in fact no design document that totally reflects the delivered code.

- **KSOS-based system generation (0%).** KSOS requires PWB/Unix for system generation. This restriction is related both to system performance and to functionality. In particular, KSOS is intended to be functionally equivalent to standard Unix (version 6), not to PWB/Unix.
- **Configurability (100%).** A complete facility to select arbitrary configurations and record them in configuration data bases exists via an automatic tool developed for KSOS. This facility allows site/distribution configuration to be distinguished from maintenance and development configuration.
- **Maintenance logistics (30%).** A document was developed to describe such logistics, but it is generally unsatisfactory.
- **Documentation (70%).** A great body of documentation exists, some of which is grossly out of date. A complete set of user's manuals exists, and is of generally high quality. Satisfactory maintenance documentation exists.

KSOS Final Report

8. ACKNOWLEDGMENTS

Acknowledgments are due the members of the KSOS project for the quality and quantity of their participation in the KSOS project. KSOS was a major milestone in secure systems technology. The technological advances of the KSOS project stand as a testament to the efforts of the entire project team.

The KSOS project was managed by Michael S. Pliner through B-Spec, by E. J. McCauley through the first half of implementation, and by Rance J. DeLong for the remainder of the implementation.

The Kernel area was initially managed by E. J. McCauley, then by Ken Biba during top level design and by Richard Neely during the detailed design and implementation. The major contributors to this area were: early design (through B-Spec) E. J. McCauley, Paul J. Drongowski, and Ken Biba; detailed design and implementation Richard Neely, John B. Nagle, and Luke C. Dion.

The Emulator area was managed by Paul J. Drongowski through B-Spec, by Rance J. DeLong through the middle of implementation and by Dan E. Ladermann and Strafford Wentworth for the remainder of the implementation. The major contributors to the Emulator area were: early design (through B-Spec) Paul J. Drongowski; detailed design (through C-Spec) Rance J. DeLong and Dan E. Ladermann; Implementation: Rance J. DeLong, Dan E. Ladermann, Glenn Skinner, and Jerry B. Scott.

The NKSr comprised of over half of the software written for KSOS. The magnitude of this effort is also reflected in the number of people that contributed to this area. The NKSr was managed by Mark Gang through the middle of the implementation phase and by Strafford Wentworth for the remainder of the project. The major contributors to the NKSr area were: early design (through B-Spec): E. J. McCauley and Mark Gang; detailed design (through C-Spec): E. J. McCauley, Mark Gang, and Strafford Wentworth; implementation: Mark Gang, Strafford Wentworth, Denise E. M. Bartels, Mark Volden, Jeremy Holden, Nanette Harter and Susan L. Dahlberg.

The Network area was managed by Norm Abramovitz through the middle of the implementation phase and by Jeremy Holden for the remainder of the project. The major contributors to this section were: Norm Abramovitz, Jeremy Holden, and Paul Fronberg.

The Verification area included writing and verifying the KSOS specifications. Many of the KSOS project members, particularly the Kernel members, assisted in writing the specifications. This area was managed by Thomas A. Berson and Lawrence Robinson. The major contributors to this area were: early specification effort (through B-Spec) Thomas A. Berson and Paul J. Drongowski; detailed specification effort Thomas A. Berson, Rich Feiertag, Lawrence Robinson, K. Larry Yelowitz, Peter G. Neumann, and Karl N. Levitt.

The Testing area was managed by Karl Kelley and Dennis C. Bunde. This area received great assistance from other KSOS team members who wrote module test frames, test scripts, etc. The major contributors to this section were: Karl Kelley, Dennis C. Bunde, Denise E. M. Bartels, and Steve Reiss.

The project team gratefully expresses, to the customer representatives from NSA and MITRE, acknowledgment and appreciation for their support and technical assistance. In particular, we would like to thank: Daniel J. Edwards (COTR, NSA), John Woodward (MITRE), Ken Shottling (NSA), Howie Weiss (NSA), Grace Nibaldi (MITRE), Susan Rajunas (MITRE), and Stan Ames (MITRE).

Acknowledgment is also due to the consultants from subcontractor, SRI International: Karl N. Levitt, Peter G. Neumann, Richard Feiertag, and Olivier Roubine.

KSOS Final Report

9. REFERENCE DOCUMENTS

9.1 KSOS Documents

"Issues in KSOS Design", (CDRL 0002BH), Ford Aerospace and Communications Corporation, Palo Alto, CA (December 1980).

"KSOS Category I Test Plan/Procedures", (CDRL 0002AR), Ford Aerospace and Communications Corporation, Palo Alto, CA (December 1980).

"KSOS Category II Test Plan/Procedures", (CDRL 0002AS), Ford Aerospace and Communications Corporation, Palo Alto, CA (December 1980).

"KSOS Change Status Report", (CDRL 0002AZ), Ford Aerospace and Communications Corporation, Palo Alto, CA (December 1980).

"KSOS Configuration Index", (CDRL 0002BA), Ford Aerospace and Communications Corporation, Palo Alto, CA (December 1980).

"KSOS Configuration Management Plan", (CDRL 0002AU), Ford Aerospace and Communications Corporation, Palo Alto, CA (December 1980).

"KSOS Distribution Unix File System", Ford Aerospace and Communications Corporation, Palo Alto, CA (February 1981).

"KSOS Implementation Plan", (CDRL 0002AC), Ford Aerospace and Communications Corporation, Palo Alto, CA (December 1980).

"KSOS Kernel Verification Results", (CDRL 0002BF), Ford Aerospace and Communications Corporation, Palo Alto, CA (December 1980).

"KSOS Maintenance and Support Plan", (CDRL 0002AE), Ford Aerospace and Communications Corporation, Palo Alto, CA (December 1980).

"KSOS Non-Kernel Security-Related Software Computer Program Development Specification (Type B5)", (CDRL 0002AH), Ford Aerospace and Communications Corporation, Palo Alto, CA (December 1980).

"KSOS Non-Kernel Security-Related Software Computer Program Product Specification (Type C5)", (CDRL 0002AX), Ford Aerospace and Communications Corporation, Palo Alto, CA (December 1980).

"KSOS Security Kernel Computer Program Development Specification (Type B5)", (CDRL 0002AF), Ford Aerospace and Communications Corporation, Palo Alto, CA (December 1980).

"KSOS Security Kernel Computer Program Product Specification (Type C5)", (CDRL 0002AV), Ford Aerospace and Communications Corporation, Palo Alto, CA (December 1980).

"KSOS Security Kernel Functional Design and Interface Specification", (CDRL 0002AQ), Ford Aerospace and Communications Corporation, Palo Alto, CA (December 1980).

"KSOS Standards and Procedures for Programs and Formal Specifications", (CDRL 0002AP), Ford Aerospace and Communications Corporation, Palo Alto, CA (December 1980).

"KSOS System Security Plan", (CDRL 0002AN), Ford Aerospace and Communications Corporation, Palo Alto, CA (December 1980).

KSOS Final Report

"KSOS System Specification (Type A)", (CDRL 0002AB), Ford Aerospace and Communications Corporation, Palo Alto, CA (November 1980).

"KSOS UNIX Emulator Computer Program Development Specification (Type B5)", (CDRL 0002AG), Ford Aerospace and Communications Corporation, Palo Alto, CA (December 1980).

"KSOS UNIX Emulator Computer Program Product Specification (Type C5)", (CDRL 0002AW), Ford Aerospace and Communications Corporation, Palo Alto, CA (December 1980).

"KSOS User's Manual", (CDRL 0002AJ), Ford Aerospace and Communications Corporation, Palo Alto, CA (December 1980).

"KSOS Verification Plan", WDL-TR7809, (CDRL 0002AD), Ford Aerospace and Communications Corporation, Palo Alto, CA (December 1980).

"KSOS Version Description Document", (CDRL 0002BB), Ford Aerospace and Communications Corporation, Palo Alto, CA (December 1980).

9.2 Other Documents

Bell, D. E. and LaPadula, L. J., "Secure Computer Systems", ESD-TR-73-278, Volume I-III, MITRE Corporation, Bedford, MA (November 1973 - June 1974).

Berson, T. A. and Barksdale, G. L. Jr., "KSOS—Development Methodology for a Secure Operating System", Proc. 1979 National Computer Conference, AFIPS Volume 48, pp. 365-371 (June 1979).

Biba, K. J., "Integrity Considerations for Secure Computer Systems", MTR-3153, MITRE Corporation, Bedford, MA (June 1975).

Boyer, R. S. and Moore, J. S., "A Computational Logic", ACM Monograph Series, Academic Press, New York, NY (1979).

Boyer, R. S. and Moore, J. S., "A Theorem Prover for Recursive Functions: A User's Manual", Report CSL-91, SRI International, Menlo Park, CA (June 1979).

Dolotta, T. A., Haight, R. C. and Mashey, J. R., "The Programmer's Workbench", Bell System Technical Journal, Vol. 57, No. 6, Part 2, pp. 2177-2200 (July-August 1978).

"Draft B5 Specifications for the MITRE Secure UNIX Prototype", Private Communication, 1977.

Feiertag, R. J., "A Technique for Proving Specifications are Multilevel Secure", Report CSL-109, SRI International, Menlo Park, CA (June 1980).

Ivie, E. L., "The Programmer's Workbench—A Machine for Software Development", CACM, Vol. 17, No. 5, pp. 746-753 (October 1977).

Kampe, M., Kline, C., Popek, G., and Walton, E., "The UCLA Data Secure UNIX Operating System", Technical Report, University of California at Los Angeles, Los Angeles, CA (July 1977).

Lampson, B., "A Note on the Confinement Problem", CACM, Vol. 16, No. 10, pp. 613-615 (October 1973).

Lipner, S. B., "A Comment on the Confinement Problem", Proc. Fifth Symposium on Operating Systems Principles, ACM SIGOPS Review, Vol. 9, No. 5, pp. 192-196 (November 1975).

KSOS Final Report

McCauley, E. J. and Drongowski, P. J. "KSOS: The Design of a Secure Operating System", Proc. 1979 National Computer Conference, AFIPS Volume 48, pp 345-353 (June 1979).

Millen, J. K., "Security Kernel Validation in Practice", CACM, Vol. 19, No. 5, pp. 243-250 (May 1976).

Parnas, D. L., "A Technique for Software Module Specification with Examples", CACM, Vol. 15, No. 5, pp. 330-336 (May 1972).

Postel, J. B., "Internetwork Protocol Specification", Version 4, Information Sciences Institute, University of Southern California, Marina del Ray, CA (September 1978).

Postel, J. B., "Specification of Internetwork Transmission Control Protocol—TCP Version 4", Information Sciences Institute, University of Southern California, Marina del Ray, CA (September 1978).

Ritchie, D. M. and Thompson, K., "The UNIX Timesharing System", CACM, Volume 17, Number 5, pp 365-375 (May 1974).

Robinson, L., Levitt, K. N., "Proof Techniques for Hierarchically Structured Programs", CACM, Vol. 20, No. 4 (April 1977).

Robinson, L., Levitt, K. N., Neumann, P. G., and Saxena, A. R., "A Formal Methodology for the Design of Operating System Software," in R. T. Yeh (ed.), *Current Trends in Programming Methodology*, Vol. 1, Prentice-Hall (April 1977).

Robinson, L., Levitt, K. N., Silverberg, A., "The HDM Handbook", Vols. I-III, SRI International, Menlo Park, CA (June 1979).

Roubine, O. and Robinson, L., *SPECIAL Reference Manual*, 3rd ed., Technical Report CSG-45, SRI International, Menlo Park, CA (January 1977).

Wirth, N., "Modula: a Language for Modular Multiprogramming", *Software Practice and Experience*, Vol 7, pp 3-35 (1977).

**DA
FILM**